HZ BOOKS
华章IT

区块链
技术丛书

巴比特

区块链底层技术和应用开发的必备用书，中国三大区块链联盟的专家联袂推荐

区块链开发指南

申屠青春　主编
宋波　张鹏　汪晓明　季宙栋　左川民　编著

机械工业出版社
China Machine Press

申屠青春

金链盟常务副秘书长，银链科技 CEO，深圳金融标准委员会会员，深圳大学博士，高级工程师，深圳市高层次人才，深圳市政府采购评审专家。曾获 2008 年深圳科技创新奖、2009 年广东省科技进步三等奖，获得发明专利授权 4 项，获 2012 年深圳发明奖。2012 年创立银链科技，2013 年开始研究区块链，2016 年转向金融行业，并发起成立金链盟。

宋波

曾在某外企支付公司任职中国区软件开发部经理，负责 ATM、EMV IC 银行卡、Kiosk 等产品，现在币信负责移动 IM 的开发工作，先后参与了钱包、算法交易、区块链、交易引擎、矿池及 APP 等产品的开发。

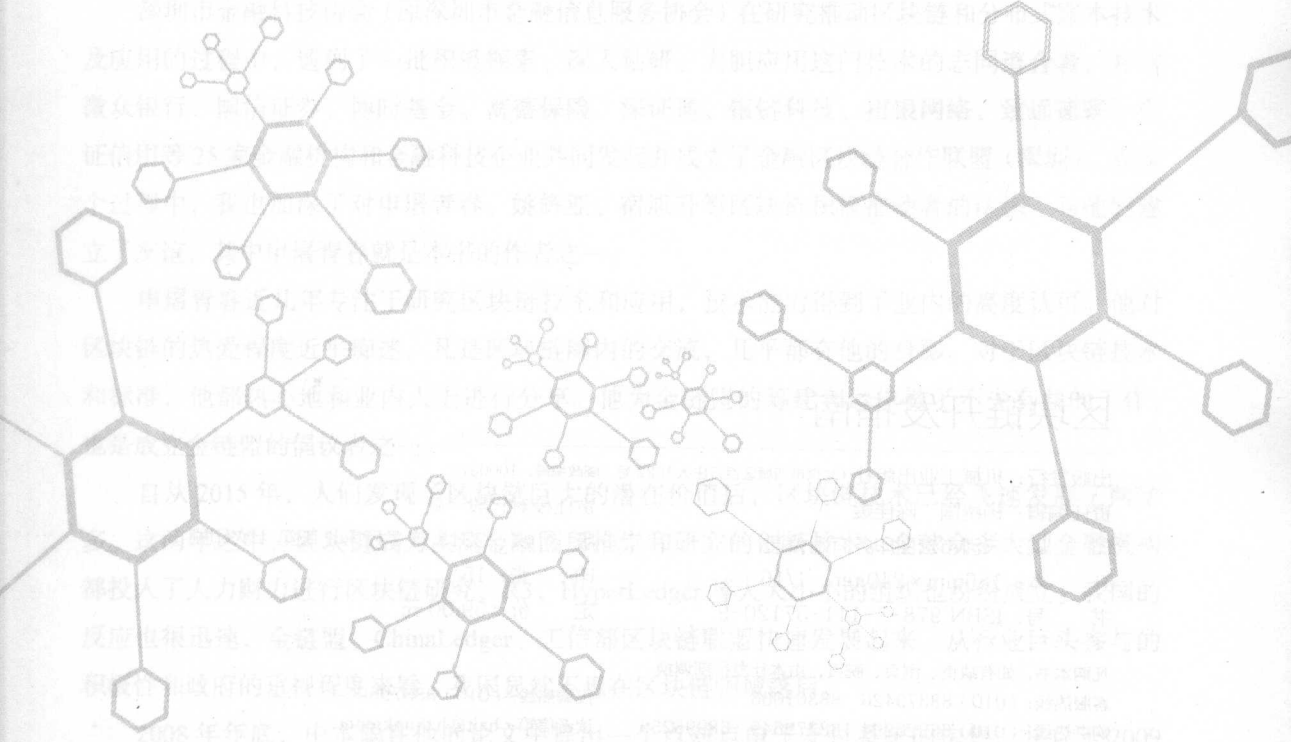
张鹏

博士，硕士生导师，现为信息工程学院讲师，中国密码学会会员。研究方向为密码学与信息安全。近年来主持或参与本领域的国家、省、市多项科技计划项目，申请国家发明专利十余项，在本领域核心学术刊物上发表文章 20 多篇，其中多篇被 SCI、EI 检索。

区块链
技术丛书

区块链
开发指南

申屠青春 主编
宋波 张鹏 汪晓明 季宙栋 左川民 编著



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

区块链开发指南 / 申屠青春主编 . —北京：机械工业出版社，2017.6（2018.2 重印）
（区块链技术丛书）

ISBN 978-7-111-57120-9

I. 区… II. 申… III. 电子商务 – 支付方式 – 指南 IV. F713.361.3-62

中国版本图书馆 CIP 数据核字（2017）第 118975 号

区块链开发指南

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）
责任编辑：杨绣国 陈佳媛
责任校对：殷 虹
印 刷：三河市宏图印务有限公司
版 次：2018 年 2 月第 1 版第 4 次印刷
开 本：186mm × 240mm 1/16
印 张：15
书 号：ISBN 978-7-111-57120-9
定 价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426 88361066
投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259
读者信箱：hzit@hzbook.com

Forward 1 推荐序一

区块链的价值实现

区块链和分布式账本技术是全球十大战略技术趋势之一，也是我国金融界、科技界过去一年高度关注的热点之一。毫无疑问，2017 年我国金融界、科技界将会加大在区块链和分布式账本技术领域的投入，同时市场上将会出现几个实际的应用。

深圳市金融科技协会（原深圳市金融信息服务协会）在研究推动区块链和分布式账本技术及应用的过程中，遇到了一批积极探索、深入钻研、大胆应用这门技术的志同道合者，并与微众银行、国信证券、博时基金、富德保险、深证通、银链科技、招银网络、致远速联、中证信用等 25 家金融机构和金融科技企业共同发起并成立了金融区块链合作联盟（深圳）。在这个过程中，我也加深了对申屠青春、姚辉亚、宿旭升等区块链积极推动者的认识，与他们建立了友谊，其中申屠青春就是本书的作者之一。

申屠青春近几年专注于研究区块链技术和应用，技术能力得到了业内的高度认可。他对区块链的热爱程度近乎痴迷，凡是区块链圈内的交流，几乎都有他的身影。对于区块链技术和标准，他都热心地和业内人士进行分享。他为金链盟的筹建和运作做了不少有益的工作，也是成立金链盟的倡议者之一。

自从 2015 年，人们发现了区块链巨大的潜在价值后，区块链技术已经飞速发展了两年多。这两年之中，区块链成为主流金融圈所推崇和研究的创新技术。全球众多大型金融机构都投入了人力财力进行区块链研究，R3、HyperLedger 等大大小小的组织也纷纷成立。我国的反应也很迅速，金链盟、ChinaLedger、工信部区块链联盟快速发展起来。从行业巨头参与的积极性和政府的重视程度来看，我国显然不想在区块链领域落后。

2008 年年底，中本聪在他的论文中提出一个点对点电子支付系统的构想，并且于 2009 年实现了比特币的原型。这个系统可以使地球上的任何人通过互联网以极致的效率进行货币交换和价值传递，无需任何第三方机构。比特币没有发明任何新技术或算法，其中涉及的技术工作量证明、时间戳、公钥体系等早已成熟。神奇的是，中本聪通过对这几项技术的组合

解决了无需可信第三方的数字资产所有权问题。从广义上讲，这些技术和思想的集合正是如今谈论的区块链。

从技术上看，区块链算是一个自由开放、没有固定形式的开源社区。众所周知，Linux 开源社区中，Linus 具有绝对权威来定制发展路线。有趣的是，区块链社区不存在这样一个角色，中本聪在 2010 年就在互联网上消失了，至今也没能确认其真实身份。也就是说，没有任何官方定义区块链该怎么实现，以及未来该怎么样发展。没有方向也许正好说明“一切皆有可能”。

去权威的社区呈现出一种百花齐放的状态，并且涌现出了大量的优秀项目和先进理念。纵览区块链的发展历史，大多创新点可归纳为共识机制、智能合约、隐私安全、可扩展性这几个方面，由于技术实现的灵活性相当大，因此更多的争论和共鸣在于设计理念和哲学上。

共识机制

中本聪在提出以工作量证明（PoW）机制作为共识算法之后，部分人认为耗能过大，于是就有了 Sunny King 设计的“环保”的股权证明（PoS）机制，后续又发展到 Bitshares 改进的股份授权证明（DPoS），并衍生出了更多的类 PoS 机制。从公有链的角度来看，共识算法就是公平和效率孰重孰轻的决策，技术实现不是难点，难点在于如何从社会学、从人性出发去设计激励机制。各种共识算法的支持者都有其合理的理由，不同共识的争论即使到现在也还一直存在。

另一个领域，金融机构的关注点在于效率、不可篡改及对应用的支持，由于不需要链上的代币激励，因而改进的拜占庭容错（PBFT）、PAXOS、RAFT 等传统分布式一致性算法就成为首选。

由此也就形成了两种区块链生态：公有链和联盟链。公有链可以任意加入，联盟链是许可加入，联盟链的用户大多是机构或公司，需要区块链契合自身的业务模式。从共识机制开始，区块链就走向了两条不同的道路，最终双方是竞争还是融合，有待后续观察。

智能合约

对智能合约的探索是出于对比特币区块链低效的脚本系统的不满，该脚本使用的是非图灵完备的堆栈语言，只能实现有限的功能。

一些智能合约研究者一直追求在区块链上运行强大的机器语言，让每个用户都能见证其运行的过程和结果，实现“程序即规则”（Code is Law）的智能环境。从图灵完备的以太坊 EVM、超级账本 ChainCode 到 Chain 平台的 ChainCore，研究者的目标是在有限的存储空间中设计一个完备合约语言和高效的底层虚拟机，甚至将传统开发语言（如 C/C++、Java）移植到

区块链上。效率和安全性的改进依然任重道远，这也是区块链领域最有技术含量的发展方向之一。

隐私安全

区块链中的个人隐私保护是强需求，特别是金融机构要使用的区块链，保护客户隐私是基本的合规条件；但同时还不能产生绝对隐私，必须要让监管者知道交易内容。

隐私安全的研究者大多需要深入掌握密码学知识，这不是一件轻松简单的工作。ZCash使用了零知识证明算法来隐藏交易双方在区块链的信息；比特币使用多输入多输出交易、隐身地址（Stealth address）和其他更多古老的混币方案来保护用户隐私。联盟链将采用数字证书认证用户，隔离一切非相关用户的数据访问。隐私安全是一把双刃剑，技术上满足隐私保护的需要，同时也增加了系统实现的复杂度；在降低透明性的同时，也要让监管更方便。

可扩展性

用户交易数的增多不可避免地会带来区块链数据膨胀的问题，可扩展性解决的是如何尽可能高效地存储不可篡改的区块链数据。业界讨论的焦点放在如下两种方向的解决方案上：

1）从交易层把部分交易迁移到子区块链上运行，即侧链、闪电网络；

2）从减少存储上着手解决，对原始数据进行裁剪分片，研究更安全的瘦客户端，只存储非全量验证数据就可正常工作。

也许从交易层进行分解可以让问题一劳永逸，但这种方法的可靠实现没有理论上那么简单。侧链技术现在正处于原型验证期，到真正实用的程度还需要一段时间。

从上述维度来看，区块链开发是一种综合能力的体现，其开发模式与互联网应用大有不同。传统互联网应用要求快速迭代，不断试错，区块链应用反而在发布前需要细致测试，对未来规划要有清晰的认识，因为一旦上线了就不是开发者能控制的：没有灰度发布、没有回滚下线、试错的成本极高。区块链是一台永不停止的信任机器，任何一次改变都需要通过共识，要明白共识的达成是极其困难的，所以在开发时一定要十分谨慎。

区块链技术的发展还面临着很多的挑战，需要更多的人才加入到探索者的队伍中。区块链开发更是需要复合型人才，分布式网络、分布式计算、密码学、编译原理、经济学等方面的内容都需要涉及，国内缺乏区块链综合技术的教程，这本书来的正当其时。本书各个主题的作者都是相关领域的专家或创业者，他们是对区块链理解最深入的一批人，具有较强的实战经验。书中各章节内容深入浅出，按时间顺序介绍区块链的技术发展，并且加入了大量的代码示例，鼓励读者动手实践，以帮助读者快速掌握区块链的开发技能，是一本值得一读的

实战型好书！

希望大家在阅读本书后有所收获。

邹　胜

2017 年 4 月

邹胜，深圳证券交易所前副总经理、深圳证券通信公司前董事长，拥有 24 年证券金融行业经验，曾领导深交所 IT 和深证通打造了第五代核心交易系统、中国证券期货业南方中心、金融云等业内领先的金融科技基础设施。现任深圳市金融科技协会联席会长，并致力于分布式交易技术在中国证券金融行业的应用推广。

Forword 2 推荐序二

区块链，推动金融代际跃升的新力量

金融为解决信息不对称而生，纵观3000年的发展历程，金融业态的变迁始终围绕着信息如何对称而展开，并基于外界环境的影响在金融化与科技化两个维度上演绎迭代。从金融化维度上看，发展主线围绕着“有中心”与“无中心”展开，哪一种业态更能解决信息对称，在不同的时空下，基于不同的金融工程技术而有不同的呈现，近400年的现代金融史正是一条从“无中心”到“有中心”再到“多中心”进而又可能回到“无中心”的演变轨迹。从科技化维度上看，信息技术的进步对金融业的影响非常敏感，人类历史上每一次信息技术的大提升都会带来金融业态的一次跳跃，尤其是近一百年来，电报、电话、海底电缆、计算机、互联网……无不带来金融业态的深刻变革，变革的指向始终是从“信息不对称”到“信息有限对称”并向着“信息对称”发展。

回顾我国改革开放后40年的金融发展，1997年无疑是至关重要的一年，那一年我国的银行业成功推出了网上银行，以此为标志，中国开始进入互联网金融时代。伴随着互联网这一千年一出的技术革命，金融业态发生着历史性的变革。不论我们以“互联网金融”还是以“金融互联网”来称谓这场变革，都不可否认金融业在解决信息对称的有效性，以及达成信息对称的效率性上，都得到了大幅提升，尤其是在以4G和智能手机为载体的移动金融出现之后，更是如此。

当然在这场长达20年的互联网金融变革中，互联网技术的内在缺陷也日益显现，网上信息的失真、可篡改、无法确权、加密强度低等特性都制约着金融业务的进一步深入，信息量越大反而越限制了信息的有效性，互联网金融开始触及自身发展的瓶颈。如何寻找新一代的替代技术解决信息的“二次不对称”，是金融业下一步演变的关键。

幸运的是，互联网本身也在迭代，并在其中一个迭代方向上出现了区块链技术。BlockChain，从诞生的第一天起就具有信息的真实、不可篡改、可确权、强加密等特征，这在某种意义上正是互联网技术的“扬弃”。而其在物理层和通信协议层与互联网的兼容，更使得

区块链技术的应用成本低、推广简便。

金融业要求信息应真实、安全、准确、权属清晰，因此我们有理由相信，互联网金融下一步演绎的方向是“区块链金融”，在区块链的路径上，继续探索哪一种“中心化形态”更有利于解决信息的对称性和效率性，从而将金融业态推向一个新的高度。

如果我们有能力预测“区块链金融”的发展轨迹，那么我们是否会形成如下这样一些观点。

- 互联网账户的区块链化。互联网金融的效应之一是让每一个金融机构都能平等地接触到终端用户，不受地域、网点、规模所限，重构大中小金融机构在金融领域重要性的自上而下排序的金字塔结构，带有典型的金融领域内的“普惠”特征。然而，这种扁平的平等触达客户的金融结构还需要另一个先决条件，那就是中小金融机构必须用技术手段自证自己的信用，自证安全可靠。显然，这有赖于区块链技术的运用。因此，在线开户、存款、支付、交易等业务整体向区块链平台迁移成为一种必然。
- 支付工具和支付体系的区块链化。以国内支付领域的现状来看，扫码支付替代磁条卡支付的趋势已经确立。然而，扫码支付的安全性始终是各方担忧的焦点。一方面，在二维码的生成与二次传播上如何增强加密强度；另一方面在二维码底层第三方支付的虚拟账户层面或银行的二类账户层面如何增强加密强度，必然会成为区块链技术发挥作用的结合点。此外，从商户收单的领域来看，商户体系只是从围绕银行展开收单和清算转变为围绕第三方支付公司展开，仍是其他“中心”的从属，基于区块链技术构建的以各商户互为中心的、以预付费卡为支付载体的“自收单体系”也将成为支付领域变革的一个重要方向。
- 征信的区块链化。互联网的持续推进引领着全球进入大数据时代，而征信在大数据时代呈现出了完全不同的业务逻辑和规则，但就目前来看，数据的权属问题将构成大数据征信模式最重大的挑战。而数据是谁的、在哪里确权、如何调用、如何计价等问题都可以借助区块链技术加以解决，从这个意义上说，大数据征信的内核是区块链征信。
- 资产证券化的区块链化。资产证券化技术成功解决了基础资产如何变成可交易的金融工具及如何计价交易的难题，从而极大地提升了金融资产的周转效率，因此被视为 20 世纪最重要的金融创新。然而，其基于线下的风险控制流程已难以适应互联网时代对金融效率和信息全面性的要求，因此，资产证券化的互联网模式已成为金融 B2B 领域的重要方向。同样，资产证券化各参与方信息的准确、不可篡改、确权、安全加密等也是 ABS 互联网化必须要面对的问题，也必然要辅助以区块链技术加以解决。
- 金融监管的区块链化。随着金融体系的日益庞大和复杂，如何监管已经成为世界难题。

2008年美国次贷危机诱发的全球金融危机，深刻反映出全球监管的滞后与漏洞。然而，随后出台的一系列监管补救法案，包括多德弗兰克法案、沃克尔法案、新巴塞尔协议等，无一例外仍在延续旧有的“规则性监管”的理念和思路，在所谓的金融杠杆控制上做出各种主观性的设置，全然没有看到在新技术运用上有突破性的理念和方法。基于后互联网时代的一系列信息技术已经完全可以做到实时、同步、自合理性设置、自预测性、自迭代的监管，即“数据性监管”，这也必然会成为下一轮全球金融监管改革的方向。区块链，基于其独特的信息处理属性，无疑会在“数据化监管”方面发挥重要的作用，有效构建监管部门、金融机构、金融客户之间的合理数据纽带。

恰如20世纪90年代末互联网进入应用开发阶段一样，区块链的产业应用同样需要一批具备区块链开发能力的人员、团队、技术组织，如何高效率地普及区块链技术、高效构建区块链技术生态至关重要。申屠青春等人编著的《区块链开发指南》，是一部难得一见的区块链实用著作，系统性地总结和提炼了区块链技术的核心属性，并从开发者的视角予以展开，相信在区块链技术和开发生态构建方面一定会发挥重要作用。也希望有更多区块链领域的技术专家和先行者贡献自身的知识和体会，共同推进区块链这一独特的信息技术，使其更迅速地与金融场景相结合，共同提升我国的金融效率与质量。

曹　彤

国金ABS云创始人

厦门国际金融技术有限公司董事长

中国区块链研究联盟副主任

推荐序三 *Foreword 3*

区块链技术的现实和未来

一直以来，科学技术都是推动时代发展的原动力。20 世纪 90 年代，随着互联网的出现，人类的信息传递方式发生了重大改变，引发了新闻媒体行业的革命，促进了电子商务的流行；移动互联网的发展带来的影响更为巨大，激发了社交的变革，带来了更为便捷、高效的包括金融服务、出行服务等在内的各类新型社会服务方式，而且社会的协作模式和运作效率从整体上也发生了深刻的变化，这些正悄然改变着社会。区块链作为近年来新兴的 IT 技术，对任何由第三方机构来进行信用背书的社会协作模式都可能会带来改变，并在金融服务、企业运作、社会生活甚至社会治理等领域引发深远的变革。

区块链是一种去中心化、去信任化的分布式账本技术，由分布式数据存储、点对点传输、共识机制、加密算法等多种技术集合而成。区块链是起源于比特币的底层技术，自 2009 年被提出以后，近年来已成为各大金融机构、IT 公司、投资机构、咨询机构关注的热点，产业界纷纷加大研发投入力度。互联网全面发展以后，已经近乎完美地解决了信息传递的问题，但是还不能自由地实现价值点到点的传递，价值的传递仍然需要中心化的可信第三方来完成，在一些应用场景中仍存在一定的局限性。区块链的出现能够在没有信任基础的双方之间建立信任，完成价值传递，因而被誉为创造信任的机器。由于其具有去中心、去信任及不可篡改的特点，区块链被认为可以应用在多种业务场景中，用来建立信任，提升透明性、可靠性与安全性。目前，区块链的应用已经不只是在数字货币和支付结算领域，在供应链金融、数字资产交易、共享经济、食品安全、慈善等多个领域均有探索，而且还将为云计算、移动互联网、物联网等新一代信息技术的发展带来新的机遇。

当前，区块链一方面带有耀眼的光环，另一方面在现实应用中还存在着很多问题亟待解决，比如：大量冗余存储、共享的数据带来了数据安全和隐私保护等方面的挑战；在去中心化、匿名的区块链系统中，使用私钥管理用户资产，私钥一旦丢失，对应的资产所有权也将丢失，而如今应用对于私钥保护基本上是用软件来实现的，理论上都存在被攻破的可能性；

另外，链上敏感数据的保护与验证也存在一定的矛盾，我们既希望重要的信息对于无关者不可见，又需要相关者在一些场景下验证信息；除此之外，智能合约也存在着一些问题，如现有司法系统对智能合约的理解和接受程度问题，部分定性合同条款难以用代码来表述的问题，代码缺陷对智能合约执行影响的问题等。璞玉亦须雕琢，对于区块链的这些问题还需要进一步探索，还有大量艰苦的工作要做。

对于区块链，业内目前有两种截然相反的态度。一种是过于乐观，看到区块链技术在比特币应用的成功之后，认为区块链技术可以很快地为社会各方面带来翻天覆地的变化。另一种态度则过于悲观，认为区块链存在的问题太多，除了比特币之外再无成功应用，且区块链可以做的工作传统信息技术完全可以解决，甚至更高效。有业内人士担心这又是一个被过度炒作的概念，最终会不了了之。从区块链技术的发展历史来看，来源于比特币的区块链技术，具有无限制加入、匿名机制、公开账本、工作量证明共识算法等技术特点，这些特点比较适合支付结算相关应用，但不具有普适性。后来为了适应不同的应用场景，在比特币平台之后，又陆续出现了多种底层平台，包括致力于打造“世界计算机”的以太坊平台、提供跨行业解决方案的 HyperLedger 项目下的 Fabric 平台、为受监管的金融行业提供专业解决方案的 R3 Corda 平台等，这些平台相互影响并不断发展。目前区块链技术除了影响力最大的比特币之外，大部分应用还处于探索阶段，成功的应用不多，但是从当前各方面的探索中，我们也看到了区块链这种去中心、去信任的价值传递网络的巨大潜力。区块链技术目前尚处在发展的初期阶段，现在最重要的是以务实的态度深入研究，特别是要吃透技术细节，结合实际场景，推动区块链相关应用扎实落地。在这方面，IT 工程师们能够发挥更加积极的作用。

虽然区块链技术仍在发展之中，仍有不少问题需要解决，但是随着基础平台的不断完善，区块链应用将得到快速发展。根据 Gartner 分析报告预测，预计经过 3 到 5 年的发展，区块链应用的落地会出现大规模的增长；未来 10 年左右，整个区块链市场将趋于成熟，广泛应用在智能合约驱动类业务、数字货币业务、机构间和机构内业务及公共记录等领域。目前，已有众多从理论和业务层面探讨区块链的图书和文章，但是技术类图书却非常稀缺。本书对于区块链的开发做了系统的介绍，是献给站在 IT 前沿开拓者的佳作。作为 IT 从业者，此时更需要把握当下，因为未来已来。让我们怀揣梦想，一起努力，共同打造更加完善的区块链服务，用科技创造美好未来！

周天虹

招商银行信息科技部总经理

2017 年 4 月

前　言 Preface

比特币于2009年诞生，在很长一段时间内，人们只知比特币，不知区块链。从2015年开始，区块链像狂风一样席卷全球，倍受金融界和科技界的关注；2015年年底，区块链技术逐渐得到国内金融界和科技界的了解和认同。

区块链行业的蓬勃发展源于区块链有可能给各行业带来巨大的变革。麦肯锡在2016年年初发布报告，指出区块链技术将在未来五年内颠覆众多行业，特别是银行业和保险业；埃森哲预测到2025年，区块链技术每年可帮助全球8大投资银行节省80亿美元至120亿美元的基础设施成本。

全球金融巨头如IBM、高盛、摩根大通、花旗银行、中国平安、瑞银、德勤、毕马威等纷纷布局区块链；区块链初创公司在全球范围内如雨后春笋般崛起，发展速度惊人。从2012年以来，全球区块链创业领域共发生207起融资/并购事件，融资额高达14亿美元。

截至2017年3月，区块链在金融业的落地应用包括跨境支付、清算结算、互助保险、电子票据、商业银行抵押品、贸易金融、数字资产登记、银行间贸易、银行间对账与审计、监管与简化流程、积分、征信、外汇交易市场、证券清算和交割等。

区块链技术还能解决供应链管理、物联网、医疗、军事、政务等领域的很多问题。例如，Warmart试图用区块链保障我国市场的猪肉供应链安全；医疗领域中，生成基于区块链的、不可更改的电子病历、检验报告等用于存证，方便解决医疗纠纷；军事防卫和信息安全化中，区块链技术可实现信息防御平台的现代化；政务中，区块链可以简化文件归档与政府公共档案管理，并且可用来发放政府社保、养老金等社会福利及居民身份存证等。

由此可见，区块链将带来一场巨大的变革。正如德勤的报告中所预言的一样：“区块链是一场改变信任的革命，将重塑金融行业。”而它作为一项伟大的技术，不仅仅对于金融行业有革新性，对于其他行业，也会有深远的影响。

而今实施“区块链+”战略所面临的最大难题是：极度缺乏从业人员。很多金融机构和企事业单位对区块链还停留在概念阶段，其开发人员不懂区块链；大部分对区块链技术感兴趣的人，或者想要从事区块链行业的技术人员，未能系统地了解区块链的原理和发展，缺乏区块链开发者应有的知识和技术储备。

为了让更多的开发人员转变成区块链开发者，让更多现有的区块链开发人员系统地理解区块链技术，在区块链领导媒体巴比特的提议和牵头下，成立了《区块链开发指南》编写小组，开始构思、编写本书。

编写小组成员有：银链科技 CEO 申屠青春、深圳大学教授张鹏、币信资深程序员宋波、朝夕网络 CEO 汪晓明、万达网络区块链研发中心总经理季宙栋、华安保险系统架构师左川民、巴比特区块链资深工程师易长军。

本书内容由申屠青春负责组织，共包含六个章节，具体分工如下：申屠青春编写第 1 章和第 2 章的大部分内容，易长军对本部分内容亦有贡献，币信的汪海波贡献了 1.4.2 节、1.4.3 节和 1.4.4 节，比特大陆的潘志彪贡献了 2.5.2 节、2.5.3 节和 2.5.4 节；张鹏编写第 3 章；宋波编写第 4 章；汪晓明编写第 5 章；季宙栋编写第 6 章的实操部分，左川民编写第 6 章的原理部分。此外，银链科技的林素兰参与第 1 章和第 2 章部分内容的编辑，万达网络的丛宏雷、张梦航参与第 6 章实操部分内容的编写。

本书以比特币、以太坊、Fabric 三种区块链的技术原理和实际操作为主要目标，全书具体内容如下。

第 1 章介绍比特币区块链，包括交易和交易链、区块和区块链、挖矿、矿池、脚本系统、合约应用案例等内容，向读者们介绍区块链基础知识。

第 2 章讲述区块链进阶技术，包括外带数据原理、Counterparty 原理、挖矿算法解析、侧链技术，以及最新的 IBLT、隔离见证、闪电网络等。

第 3 章的主要内容是区块链中使用的密码学基础，包括 Hash 函数、椭圆曲线密码体系、ECDSA 签名、Schnorr 数字签名和 Bloom filter 算法等，向开发者介绍密码学相关算法。

第 4 章是比特币区块链的编译、代码剖析、建立私链及 API 开发等实操内容。

第 5 章介绍以太坊的技术原理，包括以太坊简介、账户管理、交易原理、智能合约等，还涉及搭建私有链，智能合约开发、部署和调用等实操过程。

第 6 章介绍了 IBM 开源的区块链底层技术平台 Fabric 的原理和实操，对 Fabric 系统架构、节点、验证总账、交易背书的基本流程进行了详尽独到的分析，对 Fabric 的私有链建立和配置、链上代码的开发过程进行了详细的描述，为开发者使用 Fabric 提供技术指导。

最后，感谢编写小组各成员的配合和支持，使本书最终得以完本。感谢巴比特的李涛，

时时督促此书的编写；感谢机械工业出版社华章公司的编辑杨绣国为本书顺利出版付出的努力。编写小组期待本书能够在区块链应用开发中给开发者以参考和启发。由于成书仓促，错误之处在所难免，恳请广大读者朋友批评指正。

申屠青春

2017 年 4 月于深圳

Contents 目　录

第 1 章 Chapter 1

区块链基础

区块链究竟是什么？狭义地说，区块链就是比特币的底层技术；不过，经过 7 年的发展，区块链已经不再“依附于”比特币，而是独立地发展成为了一种革命性的技术，比特币则是区块链最大、最成功的应用。

从技术层面来看，区块链是一个基于共识机制、去中心化的公开数据库。共识机制是指在分布式系统中保证数据一致性的算法；去中心化是指参与区块链的所有节点都是权力对等的，没有高低之分，同时也指所有人都可以平等自由地参与区块链网络，唯一的限制就是个人自己的选择；公开数据库则意味着所有人都可以看到过往的区块和交易，这也保证了无法造假和改写。基于以上特性，可以总结得出：区块链由许多对等的节点组成，通过共识算法保证区块数据和交易数据的一致性，从而形成一个统一的分布式账本。

从价值层面来看，区块链是一个价值互联网，用于传递价值。目前的互联网仅用来传递消息，但是还不能可靠地传递价值；而比特币区块链却可以在全球范围内自由地传递比特币，并且能够保证不被双花、不被冒用。从这个角度来说，区块链是记录价值、传递消息和价值本身转移的一个可信账本。

这里要提一下区块链在维基百科上的官方定义：一个区块链是一个基于比特币协议的不需要许可的分布式数据库，它维护了一个持续增长的不可篡改的数据记录列表，即使对于该数据库节点的运营者们也是如此。简而言之，区块链就是区块用某种方式组织起来的链条。在区块链中，信用或记录被放在各个区块中，然后用密码签名的方式“链接”到下一个区块。这些区块在系统的每一个节点上都有完整的副本，所有的信息都带有时间戳，是可追溯的。事实上，在区块链创建之初，我们在大多数情况下谈论的区块链都是比特币的底层实现方式。

基于区块链的系统和以往的其他系统存在很多不同之处，以区块链技术为核心的系统包括如下四大最主要的特点。

- Distributed（分布式的）

  区块链是全球化的，系统上的节点是运行在太平洋某个小岛的笔记本电脑上还是运行在中国某个小镇的服务器上，对系统本身来说都是一样的，除了网络连接速度有区别之外，其他没有任何区别。区块链没有中心节点，数据分布式地存储在各个节点上，即使绝大部分节点毁灭了，只要还有 1 个节点存在，就可以重新建立并还原区块链数据。

- Autonomous（自治的）

  区块链是一种去中心化的、自治的交易体系，这种自治性表现在两个方面：1）所有节点都是对等的，每个节点都可以自由加入和离开，并且这一行为对整个区块链系统的运行没有任何影响。所有的节点都是按照相同的规则来达成共识，且无需其他节点的参与。2）区块链系统本身一旦运行起来，就可自行产生区块并且同步数据，无需人工参与。

- Contractual（按照合约执行的）

  区块链是按照合约执行的，第一体现在各个节点的运行规则（指的是交易、区块链或协议）上，按照既定的规则执行，一旦出现违背规则的行为，就会被其他节点所抛弃；第二体现在智能合约上，智能合约是一种可程序化的合同条款、规则或规定，包含在每个交易中，交易验证时必须先运行智能合约，只有通过了验证的交易才能被接受。

- Trackable（可追溯的）

  区块链的数据是公开透明的，不能被篡改，而且相关交易之间有一定的关联性，因而很容易被追溯。比如比特币区块链，每一枚比特币都有其特定的来源，通过输入可以追溯到上一个交易，或者通过输出追溯到下一个交易。

  此外，区块链代码本身也是可追溯的，区块链系统是开源软件，其对于所有的人都是公开的，因此任何人都可以查看并修改这些代码，不过修改后的代码需要经过开源社区上其他程序员的审核。

本书主要讨论区块链技术，这不仅包括了比特币区块链技术，还包含了比特币区块链所没有的一些技术，本章接下来将对区块链的一些基本知识做一个详细的介绍，包括交易和交易链、区块、挖矿、矿池、脚本、智能合约等。

1.1 交易和交易链

交易是签过名的数据结构，该数据结构会在区块链网络中广播，并被收集到区块中。它会引用以前的交易，从该交易中发送特定数量的比特币到一个或多个公钥中（即比特币

地址），并且交易未被加密（比特币体系中没有加密任何数据）。多个交易可组成一个区块（block），这些区块同样也会在区块链网络中传播，一个区块会引用上一个区块，简而言之，区块链就是由区块（block）用某种方式组织起来的链条（chain）。区块链包括成千上万个区块，而一个区块内又包含一个或多个交易，上下关联的交易组成了一个交易链，一个交易链内部可能又包含了多个交易，下面的章节将会对这些知识点进行详细解释。

1.1.1　比特币地址

比特币地址是一个由数字和字母组成的字符串，可以与任何想给你比特币的人分享。由公钥（一个同样由数字和字母组成的字符串）生成的比特币地址以数字“1”开头。下面是一个比特币地址的例子：

```
1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy
```

在交易中，比特币地址通常是以收款方的形式出现。如果把比特币交易比作一张支票，那么比特币地址就是收款人，也就是我们要写入“收款人”一栏的内容。一张支票的收款人可能是某个银行账户，也可能是某个公司、机构，甚至是现金支票。支票不需要指定一个特定的账户，而是可以用一个普通的名字作为收款人，这使得它成为一种相当灵活的支付工具。与此类似，比特币地址的使用也使比特币交易变得很灵活。比特币地址可以代表一对公钥和私钥的所有者，也可以代表其他东西。

比特币地址是由公钥经过单向的 Hash 函数生成的。用户通常所见到的比特币地址是经过“ Base58Check”编码的，这种编码使用了 58 个字符（一种 Base58 数字系统）和校验码，提高了可读性、避免了歧义，并有效地防止了在地址转录和输入中产生错误。Base58Check 编码也被用于比特币的其他地方，例如私钥、加密的密钥和脚本 Hash 中，用来提高可读性和录入的正确性。图 1-1 描述了如何从公钥生成比特币地址。

1.1.2　交易的本质

交易实质上就是包含一组输入列表和输出列表的数据结构，也可称之为转账记录，这其中就包括了交易金额、来源和收款人等信息，表 1-1 就是一个比特币交易的数据格式。

表 1-1　比特币交易数据格式

数据项	描　　述	大　　小
版本号	目前为 1	4 字节
输入数量	正整数 VI = VarInt	1 ~ 9 字节
输入列表	每块的第一个交易的第一个输入称为“coinbase”（早期版本中的内容被忽略）	<in-counter>- 许多输入
输出数量	正整数 VI = VarInt	1 ~ 9 字节
输出列表	块中的第一个交易的输出是花掉挖矿得到的比特币	<out-counter>- 许多输出
锁定时间 lock_time	如果非 0 并且序列号小于 0xFFFFFFFF，则是指块序号；如果交易已经终结，则是指时间戳	4 字节

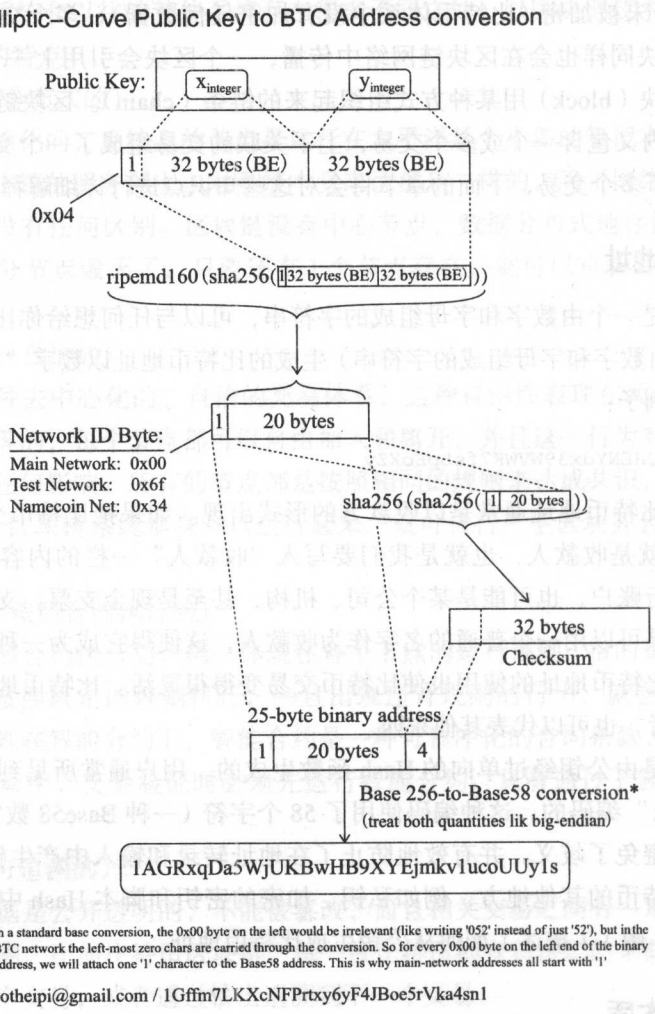


图 1-1 比特币地址生成流程

下面以一个具体的例子来说明一个区块链上的交易构成。假设有一个带有一个交易及一个输出的交易 A，其中的输入列表和输出列表如下所示：

```
Input:
Previous tx: f5d8ee39a430901c91a5917b9f2dc19d6d1a0e9cea205b009ca73dd04470b9a6
Index: 0
scriptSig: 304502206e21798a42fae0e854281abd38bacd1aeed3ee3738d9e1446618c4571d10
90db022100e2ac980643b0b82c0e88ffdfec6b64e3e6ba35e7ba5fdd7d5d6cc8d25c6b241501

Output:
Value: 5000000000
scriptPubKey: OP_DUP OP_HASH160 404371705fa9bd789a2fcd52d2c580b65d35549d
OP_EQUALVERIFY OP_CHECKSIG
```

上文表示，交易 A 的输入 0 从交易 f5d8ee39a430901c91a5917b9f2dc19d6d1a0e9cea205b009ca73dd04470b9a6 的 0 号输出中导入了 50 个比特币，然后该输出发送 50 个比特币到一个比特币地址的公钥 Hash 值（404371705fa9bd789a2fcd52d2c580b65d35549d，该公钥 Hash 值是十六进制表示，而非正常的 base58 表示）。

如果接收者想花掉这笔钱，那么他首先得创建自己的交易 B，再引用该交易 A 的 0 号输出作为交易 B 的输入。

1.1.3　输入和输出

输入是对其他交易输出的引用，一个交易中通常列有多个输入。所有被引用的输出值相加，得出的总和值会在该交易 A 的输出中用到。Previous tx 是以前交易的 Hash 值，Index 是被引用交易的特定输出号，ScriptSig 是一个脚本的前一半（脚本将在后文中详细讨论）。

脚本包含两个部分，一个签名和一个公钥，公钥属于交易输出的收款人，并且表明交易创建者允许收款人获得的输出金额；另一个部分是 ECDSA 签名，是通过对交易的 Hash 值进行 ECDSA 签名而得到的。签名和公钥一起，证明原地址的真正所有者创建了该支付交易。

输出中包含了发送比特币的指令，金额（Value）是以聪（Satoshi, 1BTC = 100 000 000 聪）为单位的数值。ScriptPubKey 是脚本的另一半（这点将在后文中详细讨论），可以有多个输出，它们共享了输入金额。一个交易中的每一个输出都只能被后来的交易当成输入引用一次。如果你不想丢币，那就需要把所有输入值的总和值发送到一个输出地址。如果输入是 50BTC，但你仅想发送 25BTC，那么比特币将创建 2 个 25BTC 的输出：一个发往目标地址，另一个则回到你的地址（称之为“找零”，详见 1.1.5 节）。在交易过程中，会产生一笔交易费，作为交易费支付的任何比特币都不能被赎回，生成这个区块的矿工将获得这笔交易费。

为了验证某个交易的输入已经被授权，可以收集被引用的输出中的所有金额。比特币体系使用了一个类似于 Forth 的脚本系统，其目的是验证从某地址发出的比特币是否真正属于该地址的拥有人，输入的 scriptSig 和被引用的输出 scriptPubKey 会按顺序运行。如果 scriptPubKey 返回真，则输入被授权，证明是地址拥有人发出了比特币。通过脚本系统，发送者可以创建非常复杂的发送条件，人们为了收到金额，首先必须满足这些条件。举个例子，可以创建一个能被任何人赎回而无需授权的输出，也可以创建一个需要 10 个不同签名的输入，或者无需公钥仅由密码赎回的输出。

1.1.4　交易类型

根据目标地址的不同，可以把交易分为如下几种类型。

（1）支付到公钥 Hash

```
scriptPubKey: OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
scriptSig: <sig><pubKey>
```

一个比特币地址只是一个 Hash 值，因而发送者无法在 scriptPubKey 中提供完整的公钥，

当要赎回比特币时，接收者需要同时提供签名 scriptSig 和公钥 scriptPubKey，脚本系统会验证公钥的 Hash 值与 scriptPubKey 中的 Hash 值是否匹配，同时还会检查公钥和签名是否匹配。检查过程见 4.2.5 节。

（2）支付到脚本 Hash

该类交易非常有意义，未来应该会在某些场合频繁使用。该类交易的接受地址不是通常意义上的地址，而是一个多签地址，以 3 开头。比如，三对公钥对可以生成一个多签地址。需要在生成过程中指定 n of 3 中的 n，n 的范围是 [1, 3]，若 n = 1，则仅需要一个私钥签名即可花费该地址的币，若 n = 3，则需要三把私钥依次签名才可以。

地址以 3 开头，可以实现多方管理资产，极大地提高安全性，也可以轻松实现基于比特币原生的三方交易担保支付。一个 m-of-n 的模式如下：

```
m {pubkey}...{pubkey} n OP_CHECKMULTISIG
```

其中，m 和 n 需要满足：$1 \leqslant n \leqslant 20$，$m \leqslant n$。

m 和 n 可以是 1 of 1、1 of 2、2 of 3 等组合，通常选择 n = 3。

- 1 of 3，最大程度私钥冗余。防丢私钥损失，3 把私钥中任意一把即可签名发币，即使丢失 2 把也可以保障不受损失。
- 2 of 3，提高私钥冗余度的同时解决单点信任问题。3 把私钥中的任意 2 把私钥可签名发币，对于三方不完全信任的情形，即中介交易，非常适用。
- 3 of 3，最大程度解决资金信任问题，无私钥冗余。必须 3 把私钥全部签名才能发币，适用于多方共同管理的重要资产，但是任何一方遗失私钥均会造成严重损失。

多签地址的交易构造、签名、发送过程与普通交易类似。

（3）挖矿交易

挖矿（coinbase）交易用于凭空产生比特币。挖矿交易只有一个输入，该输入有一个“coinbase”参数，没有 scriptSig，“coinbase”中的数据可以是任意内容，它不会被使用。比特币把压缩的当前目标 Hash 值和任意精确度的“extraNonce”都存储在这儿，区块头中的 Nonce 每次一溢出，它们就会增长。extraNonce 有助于扩大工作量证明函数的范围，矿工很容易修改 Nonce（4 字节）、时间戳和 extraNonce（2 ~ 100 字节）。

挖矿交易的输出金额在一段时间内是固定值，初始是 50 个比特币，每 21 万个区块后减半，目前已经经历了两次减半，因而是 12.5 个比特币。输出地址可以是任何地址，一般是矿工或矿池的比特币地址。

Nonce 溢出是指在对一个块进行散列时，Nonce 从 0 开始，每计算一次 Hash 都要增长一次，因而有可能会出现超过数值范围的情况，这时，extraNonce 就要相应增长以存储 Nonce。

1.1.5 找零地址

在实际的区块链交易中，假设 A 拥有一个比特币地址，里面包含着还没有花费过的 10

个比特币。B 也有一个比特币地址，里面一分钱也没有。当 A 想向 B 支付 10 个比特币时，A 地址里的未花费输出变为零，而 B 的则会变为 10 BTC。如果 A 想支付的金额与所拥有的相同，自然不会存在需要找零钱的问题。不过当手头的金额比要支付的大时，找零自然也是天经地义的事情。

假设 A 的地址上有 35 个比特币（如图 1-2 所示），当 A 想向 B 支付 8 个比特币时，如图 1-2 所示，只需要使用包含着 20 个比特币的那一笔未消费支出，并设置好要支持的金额即可，剩下的 12 个比特币则会返回给 A，以便 A 在将来可以继续使用。

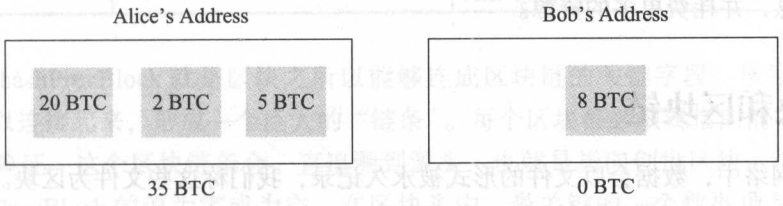


图 1-2　找零示意图

这样就有了一个找零机制，实际上，比特币在交易时会把消费时所用的地址（消费地址）的余额设置为零。当需要支付的金额小于可用余额时，在交易信息中必须告诉比特币网络零钱将要被发送至哪个地址，即“找零地址”。找零地址可能是也可能不是原先的发送地址。除此之外，发送地址所留下的剩余款项将由网络作为交易费支付给矿工。在上面的例子里，A 可以选择将找回的零钱发送到一个新创建的找零地址上，或者将原先发送的地址设置为找零地址，并将零钱返回。虽然将发送地址作为找零地址对 A 而言是方便了管理，不过这也可能会造成 A 的隐私性降低，在一定程度上还可能会影响到 B 的匿名性。

根据设计，每一笔比特币交易将在一个称为“区块链”的全球性的公共总账上永久可见，这就意味着任何人随时都可以在上面进行跟踪查询。通过将某个比特币地址与其使用者关联起来，好事者都可以据此绘制关于这个人与他人之间的资金转移的关系图。但如果是将找回的零钱发送至一个新创建的地址，那么就可以让这种追踪变得更加困难。

要理解这一点，可以参考图 1-3。假设从地址 A 发送比特币到地址 B 后，零钱返回地址为 A，则区块链会揭示地址 A 向地址 B 支付了一笔钱。同样的道理，如果有两个或两个以上地址参与其中，任何涉及这个接收零钱的找零地址都会揭示 A 作为支付方的交易。假如某个控制着的任何接收地址或付款地址的人其身份是众所周知的，那么其他有过交易往来的各方的身份也有可能被推断出来。

现在想象一下，地址 A 发起付款到地址 B，但此时将找零地址更改为新生成的地址 C，如图 1-4 所示。如果没有进一步的信息，那么其他人所能知道的，只是有一个交易拆分了地址 A 的余额至地址 B 和 C。而地址 B 或 C 的主人可能是也可能不是 A。由于新地址 C 的加入，让整个交易的真相变得更加扑朔离迷：哪一个地址代表着被支付方，哪一个地址代表着找回的零钱呢？

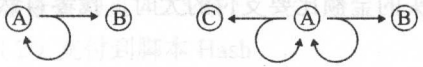


图 1-3 两种找零方案

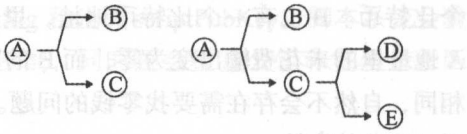


图 1-4 重新生成找零地址

当所有各方都将零钱发送至新创建的地址时，要想将个人身份与地址相关联，就必须收集更多的信息，并耗费更多的资源。

1.2 区块和区块链

比特币网络中，数据会以文件的形式被永久记录，我们称这些文件为区块。一个区块是一些或所有最新比特币交易的记录集，且未被其他先前的区块记录。可以将区块想象为一个城市记录者其记录本上单独的一页纸（对房地产产权的变更记录），或者是股票交易所的总账本。在绝大多数情况下，新区块会被加入到记录的最后（在比特币中的名称为区块链），一旦写上，就再也不能改变或删除。每个区块记录了它被创建之前发生的所有事件。

1.2.1 区块结构

一个区块的结构如表 1-2 所示。

表 1-2 区块结构示意图

数据项	描　述	长　度
Magic no（魔法数）	总是 0xD9B4BEF9	4 字节
Blocksize（区块大小）	到区块结束的字节长度	4 字节
Blockheader（区块头）	包含 6 个数据项	80 字节
Transaction counter（交易数量）	正整数 VI = VarInt	1 ~ 9 字节
Transactions（交易）	交易列表（非空）	<Transaction counter>- 许多交易

每个区块都包括了一个被称为“魔法数”的常数 0xD9B4BEF9、区块的大小、区块头、区块所包含的交易数量及部分或所有的近期新交易。在每个区块中，对整个区块链起决定作用的是区块头，如表 1-3 所示，接下来本章将会对每一个字段都做出比较详细的解释。

表 1-3 区块头描述

数据项	目　的	更新时间	大小（字节）
Version（版本）	区域版本号	更新软件后，它指定了一个新的版本号	4
hashPrevBlock（前一区块的 Hash）	前一区块的 256 位 Hash 值	新的区块进来时	32

（续）

数据项	目　　的	更新时间	大小（字节）
hashMerkleRoot Merkle（根节点 Hash 值）	基于一个区块中所有交易的 256 位 Hash 值	接受一个交易时	32
Time（时间戳）	从 1970-01-01 00:00 UTC 开始到现在，以秒为单位的当前时间戳	每几秒就更新	4
Bits（当前目标 Hash 值）	压缩格式的当前目标 Hash 值	当挖矿难度调整时	4
Nonce（随机数）	从 0 开始的 32 位随机数	产生 Hash 时（每次产生 Hash 随机数时都要增长）	4

这里的 hashPrevBlock 就是区块之所以能够连成区块链的关键字段，该字段使得各个区块之间可以连接起来，形成一个巨大的“链条”。每个区块都必须要指向前一个区块，否则无法通过验证。这个区块链条会一直追溯到源头，也就是指向创世区块。很显然，创世区块的 hashPrevBlock 的值为零或为空。在区块头中，最关键的一个数据项是一个随机数 Nonce，这串数字是一个答案，而这个答案对于每一个区块来说都是唯一的，它的特点具体如下。

- ❑ 这个答案很难获得。
- ❑ 有效答案有多个，不过我们只需要找到一个答案就可以了。
- ❑ 其他节点对有效答案的验证很容易。

正是因为问题很难解答，没有固定的算法可以求出答案，所以唯一的做法就是不断尝试，找寻这个答案的做法就是“挖矿”，可以想象，会有很多人同时都在“挖矿”，他们之间是相互竞争的关系。

区块内包含许多交易，它们通过 Merkle 根节点间接被散列，以保证矿工能及时追踪一个正在打包的区块内交易的变化情况。一旦生成 Merkle 根节点，那么对包含一个交易的区块做散列所花的时间，与对包含 1 万个交易的区块做散列所花的时间是一样的。

目标 Hash 值的压缩格式是一个特殊的浮点编码类型，首字节是指数（仅使用了 5 个最低位），后 3 个字节是尾数，它能表示 256 位的数值。一个区块头的 SHA-256（一种单向函数的算法，可形成长度为 256 位的串）值必定要小于或等于目标 Hash 值，该区块才能被网络所接受。目标 Hash 值越低，产生一个新区块的难度就越大。

Merkle 树是 Hash 的二叉树。在比特币中会两次使用 SHA-256 算法来生成 Merkle 树，如果叶子个数为奇数，则要重复计算最后一个叶子的两次 SHA-256 值，以达到偶数叶子节点的要求。

计算过程：首先按照区块中交易的两次 SHA-256 进行散列，然后按照 Hash 值的大小进行排序，生成最底层。第二层的每个元素则是相连续的两个 Hash 值的两次 SHA-256 的 Hash 值。之后，会重复这个过程，直到某一层只有一个 Hash 值为止，这就是 Merkle 根。举例来说，想象有 3 个交易，a、b、c，那么 Merkle 根的生成过程如下所示：

```
d1 = dhash(a)
d2 = dhash(b)
d3 = dhash(c)
d4 = dhash(c)          # 只有3个元素，是奇数，因而将最后一个元素重算一次
d5 = dhash(d1 concat d2)
d6 = dhash(d3 concat d4)
d7 = dhash(d5 concat d6)
```

这里的d7就是以上三个交易的Merkle根。需要注意的是，Merkle树的Hash值是小头位序（即高位在后，是数字在计算机中的一种表示形式）。对于某些实现和计算来说，在散列计算前应该先按位反转，之后在散列计算后再反转一次。

1.2.2 创世块

创世块（Genesis Block）是指区块链的第一个区块，现在的比特币客户端版本把创世区块号定为0，以前的版本把该区块号定为1。以下是创世块的一种表示形式，它出现在以前的比特币代码的注释中，第一个代码段定义了创建该块所需要的所有变量，第二个代码段是标准的区块类格式，还包含了第一个代码段中缩短版本的数据。

```
GetHash()= 0x000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
hashMerkleRoot = 0x4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b
txNew.vin[0].scriptSig    = 4866047994
0x736B6E616220726F662074756F6C69616220646E6F63657320666F206B6E697262206E6F20726
F6C6C65636E6168432039303032 2F6E614A2F33302073656D695420656854
txNew.vout[0].nValue    = 5000000000
txNew.vout[0].scriptPubKey =
0x5F1DF16B2B704C8A578D0BBAF74D385CDE12C11EE50455F3C438EF4C3FBCF649B6DE611FEAE06
279A60939E028A8D65C10B73071A6F16719274855FEB0FD8A6704 OP_CHECKSIG
block.nVersion = 1
block.nTime    = 1231006505
block.nBits    = 0x1d00ffff
block.nNonce = 2083236893

CBlock(hash=000000000019d6, ver=1, hashPrevBlock=00000000000000, hashMerkleRoot=
4a5e1e, nTime=1231006505, nBits=1d00ffff, nNonce=2083236893, vtx=1)
  CTransaction(hash=4a5e1e, ver=1, vin.size=1, vout.size=1, nLockTime=0)
    CTxIn(COutPoint(000000, -1), coinbase 04ffff001d0104455468652054696d65732030
332f4a616e2f32303039204368616e63656c6c6f72206f6e206272696e6b206f66207365636f6e64206
261696c6f757420666f722062616e6b73)
    CTxOut(nValue=50.00000000, scriptPubKey=0x5F1DF16B2B704C8A578D0B)
  vMerkleTree: 4a5e1e
```

coinbase参数（看上面的十六进制）中包含了“The Times 03/Jan/2009 Chancellor on brink of second bailout for banks.”这句话。

这句话翻译过来就是“2009年1月3日，首相第二次对处于崩溃边缘的银行进行紧急救助”，这句话正是泰晤士报当天的头版文章标题（如图1-5所示）。这应该是一个该区块在

2009 年 1 月 3 日或之后创建的一个证据，同时也是对银行系统采用部分准备金制度导致不稳定性的一个说明。

图 1-5　2009 年 1 月 3 日的泰晤士报

创世块 50BTC 的收益被发送到如下地址：1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa，我们称该交易为创世交易。

创世块的收益花不掉，原因如下：比特币客户端把区块和交易分别存储在两个数据库中，当客户端发现区块数据库为空时，就会用代码直接生成一个创世块，但是没有把创世交易存储到客户端的交易数据库中，比特币网络一旦收到要花掉创世交易输出的交易时，因为在交易数据库中找不到创世交易，因而都会拒绝，也就是说花不掉这 50 个币了。出现这种情况很可能是中本聪为了纪念创世交易，故意而为的。

创世块的数据结构如下所示。

- 01000000：指版本号。
- 0000000000000000000000000000000000000000000000000000000000000000：为 prev block。
- 3BA3EDFD7A7B12B27AC72C3E67768F617FC81BC3888A51323A9FB8AA4B1E5E4A：为 Merkle 根。

- 29AB5F49：时间戳。
- FFFF001D：目标 Hash 值。
- 1DAC2B7C：随机数。
- 01：交易个数。
- 01000000：版本。
- 01：输入个数。
- 0000000000000000000000000000000000000000000000000000000000000000FFFFFFFF：前一个输出。
- 4D：脚本长度。
- 04FFFF001D0104455468652054696D65732030332F4A616E2F32303039204368616E63656C6C6F72206F6E206272696E6B206F66207365636F6E64206261696C6F757420666F722062616E6B73：scriptsig 脚本。
- FFFFFFFF：序列号。
- 01：输出个数。
- 00F2052A01000000：50 BTC 的收益。
- 43：指脚本 scriptPubKey 的长度。
- 4104678AFDB0FE5548271967F1A67130B7105CD6A828E03909A67962E0EA1F61DEB649F6BC3F4CEF38C4F35504E51EC112DE5C384DF7BA0B8D578A4C702B6BF11D5FAC：脚本 scriptPubKey。
- 00000000：锁定时间。

JSON 版本的创世块如下所示：


```
{
  "hash":"000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f",
  "ver":1,  "prev_block":"0000000000000000000000000000000000000000000000000000000000000000",
"mrkl_root":"4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b",
  "time":1231006505,
  "bits":486604799,
  "nonce":2083236893,
  "n_tx":1,
  "size":285,
  "tx":[
    {
      "hash":"4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b",
      "ver":1,
      "vin_sz":1,
      "vout_sz":1,
      "lock_time":0,
      "size":204,
      "in":[
        {
          "prev_out":{
```

```
    "hash":"0000000000000000000000000000000000000000000000000000000000000000",
                "n":4294967295
              },
            "coinbase":"04ffff001d0104455468652054696d65732030332f4a616e2f3230303920
4368616e63656c6c6f72206f6e206272696e6b206f66207365636f6e64206261696c6f757420666f722
062616e6b73"
            }
          ],
          "out":[
            {
              "value":"50.00000000",
            "scriptPubKey":"04678afdb0fe5548271967f1a67130b7105cd6a828e03909a67962e
0ea1f61deb649f6bc3f4cef38c4f35504e51ec112de5c384df7ba0b8d578a4c702b6bf11d5f OP_CHECKSIG"
            }
          ]
        }
      ],
      "mrkl_tree":[
        "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b"
      ]
    }
```


1.2.3　区块链原理

区块链是所有比特币节点共享的交易数据库，这些节点基于比特币协议参与到比特币网络中来。区块链包含每一个曾在比特币系统执行过的交易，根据这个信息，人们可以找到任何时候任一个地址中的币数量。

由于每个区块包含前一个区块的 Hash 值，这就使得从创世块到当前块形成了一条块链，每个区块必定按时间顺序跟随在前一个区块之后，区块链结构如图 1-6 所示。因为不知道前一块区块的 Hash 值，因此没法生成当前区块，所以要改变一个已经在块链中存在了一段时间的区块，从计算上来说是不可行的，如果它被改变，那么它之后的每个区块都必须随之改变。这些特性使得双花比特币非常困难，区块链是比特币的最大创新。

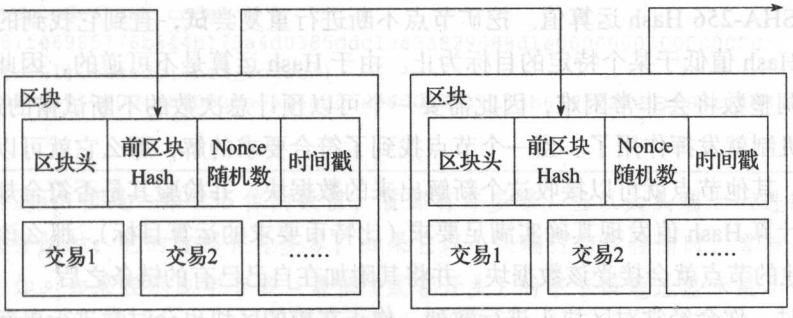


图 1-6　区块链示意图

如果一个区块是最长块链的最后一个区块，那么诚实的矿工只会在这个区块的基础上生

成后续块（创建新区块时通过引用该区块来实现）。“长度”是指被计算成区块链的所有联合难度，而不是区块的数量，尽管这个区别仅仅在防御几个潜在攻击时有用。如果一个区块链中的所有区块和交易均有效，则该区块链有效，并且要以创世块开头。

对于区块链中的任何区块来说，只有一条通向创世块的路径。然而，从创世块出发，却可能有分叉。当两个区块产生的时间仅相差几秒时，可能会产生包含一个区块的分叉。当出现以上现象时，矿工节点会根据收到区块的时间，在先收到的区块的基础上继续挖矿。哪个区块的后续区块先出现，那么这个区块就会被包括进主链，因为这条块链更长。

短区块链（或有效区块链）中的区块没有作用，当比特币客户端转向另一个长区块链时，短区块链中所有有效的交易都将被重新加入到交易队列池中，并被包括到另一个区块中。短区块链中的区块收益不会在长链中出现，因而这些收益实际上是丢失了，这就是比特币网络设定 100 个区块成熟时间的原因。

短区块链中的区块经常被称为“孤立”区块，事实上这些区块都有父区块，并且可能还有子区块，只不过这些区块链未被包含进比特币主链，就好像被孤立了一样。

1.3 挖矿、矿池

1.3.1 挖矿原理与区块的产生

比特币的挖矿和节点软件是基于对等网络、数字签名来发起和验证交易的。节点向网络广播交易，这些广播出来的交易需要经过矿工的验证，矿工们会用自己的工作证明结果来表达确认，确认后的交易会被打包到数据块中，数据块会串起来形成连续的数据块链。中本聪本人设计了第一版的比特币挖矿程序，这一程序随后被开发为广泛使用的第一代挖矿软件 bitcoind，这一代软件在 2009 年到 2010 年期间都比较流行。

每一个比特币的节点都会收集所有尚未确认的交易，并且会将其归集到一个数据块中，这个数据块将和前面一个数据块集成在一起。矿工节点会附加一个随机调整数，并计算前一个数据块的 SHA-256 Hash 运算值。挖矿节点不断进行重复尝试，直到它找到的随机调整数使得产生的 Hash 值低于某个特定的目标为止。由于 Hash 运算是不可逆的，因此寻找到符合要求的随机调整数将会非常困难，因此需要一个可以预计总次数的不断试错的过程。这时，工作量证明机制就发挥作用了。当一个节点找到了符合要求的解，那么它就可以向全网广播自己的结果。其他节点就可以接收这个新解出来的数据块，并检验其是否符合规则。只要其他节点通过计算 Hash 值发现其确实满足要求（比特币要求的运算目标），那么该数据块就是有效的，其他的节点就会接受该数据块，并将其附加在自己已有的链条之后。

当挖矿时，你会经常对区块头进行散列，你正在挖的区块也会时常进行更新，一个区块头如上文所述的表 1-3 所示。

表 1-3 中的大部分数据项对所有用户都是一致的，不过，在时间戳上可能会有些区别。

如果当前区块的时间戳大于前 11 个区块的平均时间戳，并且小于“网络调整时间（Network-Adjusted Time）”+ 2 小时，则认为该时间戳是有效的。其中的“网络调整时间”是指与你相连接的所有节点的平均时间。当节点 A 连接到节点 B 时，A 将从 B 处得到一个 UTC 的时间戳，A 先将其转换成本地 UTC 并保存起来，网络调整时间等于所有节点的本地 UTC 时间 + 所有相连节点的偏移量平均值，然而，该网络时间永远不会调整到超过本地系统时间 70 分钟以上。

Nonce 随机数通常都不会相同，但是它以严格的线性方式在增长，从 0 开始，每次执行散列时都会增长，当 Nonce 溢出时（此事经常发生），挖矿交易的 extraNonce 项就会增长，其将改变 Merkle 树的根节点。

假定针对这些数据项，人们经常会独自产生同样序列号的 Hash 值，那么，最快的矿机通常会赢。然而，两人产生同样的 Merkle 根节点基本上（或几乎）是不可能的，因为区块中的第一个交易是挖矿交易并且会“发送”到你独一无二的比特币地址上。而你的区块与其他人的区块是有区别的，因此，产生的 Hash 值肯定也会（几乎可以肯定）不同，你计算的每个 Hash 值和网络中的其他人一样，都有同样的获胜机会。

比特币使用 SHA256（SHA256（区块头））计算 Hash 值，但要注意字节序。例如，以下的 Python 代码用于计算某一区块的 Hash 值，使用的是 2011 年 6 月区块号为 125552 的最小 Hash 值。该区块头建立在表 1-3 所示的 6 个数据项之上，并且以十六进制的小端结尾方式连接在一起。

```
>>> import hashlib
>>> header_hex = ("01000000" +
    "81cd02ab7e569e8bcd9317e2fe99f2de44d49ab2b8851ba4a308000000000000" +
    "e320b6c2fffc8d750423db8b1eb942ae710e951ed797f7affc8892b0f1fc122b" +
    "c7f5d74d" +
    "f2b9441a" +
     "42a14695")
>>> header_bin = header_hex.decode('hex')
>>> hash = hashlib.sha256(hashlib.sha256(header_bin).digest()).digest()
>>> hash.encode('hex_codec')
  '1dbd981fe6985776b644b173a4d0385ddc1aa2a829688d1e0000000000000000'
>>> hash[::-1].encode('hex_codec')
  '00000000000000001e8d6829a8a21adc5d38d0a473b144b6765798e61f98bd1d'
```

注意　实际的 Hash 值是一串 256 位的数值，首部有许多零。当以大头端十六进制常数方式打印或存储时，它的首部有许多零；如果它以小头端打印或存储时，零就会变换到尾部。例如，如果表示成字节串 – 最低（或者开头）的字节串地址显示最小位的数，那么这就是小头端表示。blockexplorer 的输出把 Hash 值显示为大头端表示的数值，因为数字的表示通常是首部数字是最大的数字（从左向右读）。

1.3.2 挖矿难度

挖矿难度是对挖矿困难程度的度量，即指计算符合给定目标的一个 Hash 值的困难程度。比特币网络有一个全局的区块难度，有效的区域必须有一个 Hash 值，该 Hash 值必须小于给定的目标 Hash 值。矿池也会有一个自定义的共享难度，用来设定产生股份的最低难度限制。

难度每过 2016 块就会改变一次，计算公式为：

$$\text{difficulty} = \text{difficulty\_1\_target}/\text{current\_target}$$

其中，目标（target）是一个 256 位长的数值。

测量难度有许多不同的方法，通过这些方法得到的 difficulty_1_target 有可能也会不同。传统情况下，它表示一个 Hash 值，前 32 位为 0，后续部分为 1（称之为矿池难度或 pdiff），比特币协议把目标 Hash 值表示成一个固定精度的自定义浮点类型，因而，比特币客户端用该值来估计难度（称之为 bdiff）。

难度经常被存储在区块中，每个块存储一个十六制的目标 Hash 值的压缩表达式（称之为 Bits），目标 Hash 值可以通过预先定义的公式计算出来。例如：如果区块中压缩的目标 Hash 值为 0x1b0404cb，那么十六进制的目标 Hash 值就为如下形式：

$$0x0404cb \times 2^{8\times(0x1b-3)} = 0x00000000000404CB000000000000000000000000000000000000000000000000$$

因而目标 Hash 值为 0x1b0404cb 时，难度为：

0x00000000FFFF0000000000000000000000000000000000000000000000000000/
0x00000000000404CB000000000000000000000000000000000000000000000000
= 16307.420938523983 (bdiff)

或者：

0x00000000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF/
0x00000000000404CB000000000000000000000000000000000000000000000000
= 16307.669773817162 (pdiff)

其中：0x00000000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF 是挖矿机使用的最大目标 Hash 值。而 0x00000000FFFF0000000000000000000000000000000000000000000000000000 则是比特币网络使用的浮点编码类型，后面的位数被缩短了。

下面是一个快速计算比特币难度的方法，这里使用的算法是修改的泰勒序列（读者可以参考 wikipedia 上的教程），并且依赖记录来转换难度计算。

```
#include <iostream>
#include <cmath>
inline float fast_log(float val)
{
  int * const exp_ptr = reinterpret_cast <int *>(&val);
  int x = *exp_ptr;
  const int log_2 = ((x >> 23) & 255) - 128;
  x &= ~(255 << 23);
```

```
    x += 127 << 23;
    *exp_ptr = x;

    val = ((-1.0f/3) * val + 2) * val - 2.0f/3;
    return ((val + log_2) * 0.69314718f);
}

float difficulty(unsigned int bits)
{
    static double max_body = fast_log(0x00ffff), scaland = fast_log(256);
    return exp(max_body - fast_log(bits & 0x00ffffff) + scaland * (0x1d - ((bits
& 0xff000000) >> 24)));
}

int main()
{
    std::cout << difficulty(0x1b0404cb) << std::endl;
    return 0;
}
```

如果想要了解难度计算的数学原理，那么可以看看如下的 Python 代码：

```
import decimal, math
l = math.log
e = math.e
print 0x00ffff * 2**(8*(0x1d - 3)) / float(0x0404cb * 2**(8*(0x1b - 3)))
print l(0x00ffff * 2**(8*(0x1d - 3)) / float(0x0404cb * 2**(8*(0x1b - 3))))
print l(0x00ffff * 2**(8*(0x1d - 3))) - l(0x0404cb * 2**(8*(0x1b - 3)))
print l(0x00ffff) + l(2**(8*(0x1d - 3))) - l(0x0404cb) - l(2**(8*(0x1b - 3)))
print l(0x00ffff) + (8*(0x1d - 3))*l(2) - l(0x0404cb) - (8*(0x1b - 3))*l(2)
print l(0x00ffff / float(0x0404cb)) + (8*(0x1d - 3))*l(2) - (8*(0x1b - 3))*l(2)
print l(0x00ffff / float(0x0404cb)) + (0x1d - 0x1b)*l(2**8)
```

前难度可以通过 http://blockexplorer.com/q/getdifficulty 来获得，下一个难度可以通过 http://blockexplorer.com/q/estimate 来获得。难度的变化情况可以查看 http://bitcoin.sipa.be/。

最大难度大约等于 maximum_target/1（因为 0 会导致无穷大），这是一个非常大的数值，大约为 2^{224}；当 maximum_target 为最小值 1 时，则最小难度值也为 1。

难度可根据以前 2016 个区块的产生时间而定，每过 2016 块就会改变一次。预计每隔 10 分钟产生一个区块，因而产生 2016 个区块要花费 2 周时间。如果前 2016 个区块的产生时间多于两周，则难度会降低；否则难度就会增加。

为了找到新区块，该区块的 Hash 值必须小于目标 Hash 值，实际上它是一个在 0 到 $2^{256}-1$ 之间的随机数，难度 1 的偏移量是：

$$0xffff \times 2^{208}$$

难度 D 的偏移量是：

$$(0xffff \times 2^{208})/D$$

在难度 D 下，为了找到新区块，预期要计算的 Hash 数量是：

$$D \times 2^{256}/(0xffff \times 2^{208})$$

或者只是：

$$D \times 2^{48}/0xffff$$

难度的设定，是为了以每 10 分钟一个区块的速度产生 2016 个区块，因而，在 600 秒内计算（$D \times 2^{48}/0xffff$）个 Hash，这就意味着产生 2016 个区块的网络 Hash 速率（算力）是：

$$D \times 2^{48}/0xffff/600$$

可以进一步简化为：

$$D \times 2^{32}/600$$

以上公式有较好的精度。

在难度为 1 时，算力是 7Mhashes/ 秒，难度是 5 006 860 589，这就意味着以前 2016 个区块被找到，其平均算力是：35.840PHash/s。

$$5\ 006\ 860\ 589 \times 2^{32}/600 \approx 35.840\ \text{PHash/s}$$

发现一个区块的平均时间，可以用以下公式估计：

$$\text{时间} = \text{难度} \times 2^{32}/\text{算力}$$

其中，难度是当前的难度，算力是指矿机的计算能力，以 hashes/s 为单位，时间是你找到两个区块的平均时间。举例来说，使用 Python 计算，算力为 1Ghashes/s 的矿机，难度在 20 000 时，产生一个新区块的时间，计算式如下：

```
$ python -c "print 20000 * 2**32 / 10**9 / 60 / 60.0"
23.85
```

其中 ** 表示指数，该语句运算的结果就是：找到一个新区块要花费近 1 天的时间。

1.3.3 矿池原理与商业模式

随着生成区块的难度逐步增加，挖矿变成了一个碰运气的事情，单一节点要生成一个区块需要花费数年的时间（除非这个单一节点拥有大量的计算力）。为了激励计算力较低的用户继续参与挖矿，矿池就出现了。在一个矿池里，许多不同的人贡献出自己的计算力来生成一个区块，然后再根据每个人的贡献比例来分发奖励。通过这种方式，要得到那个 50 个比特币的奖励就不必等待数年的时间了，小矿工能定期得到属于他们那部分的比特币奖励。一个 share（贡献 / 股份）为一个矿池给客户端的一个合法的工作证明，同时这也是用来生成区块的工作证明，但是没有这么复杂，只需要很少的时间就能达到一个 share。

矿池是比特币（Bitcoin）等 P2P 密码学虚拟货币开采所必需的基础设施，一般是对外开放的团队开采服务器。其存在的意义是提升比特币开采的稳定性，使矿工薪酬趋于稳定，目前国内较为著名的比特币商业矿池有 F2Pool、BTCC Pool、BW Pool、BTC.com 等。

关于矿池挖矿的方式，目前存在如下几种不同的方式。

- Slush 方式：Slush 矿池基于积分制，较老的 shares 将比新的 shares 拥有更低的权重，以减少一轮中切换矿池的投机分子。

- Pay-Per-Share 方式：该方式为立即为每一个 share 支付报酬。该支出来源于矿池现有的比特币资金，因此可以立即取现，而不用等待区块生成完毕或确认之后。这样可以避免矿池运营者幕后操纵。这种方法减少了矿工的风险，但将风险转移给了矿池的运营者。运营者可以收取手续费来弥补这些风险可能造成的损失。
- Luke-Jr 方式：该方式借用了其他方式的长处，如 Slush 方式一样，矿工需要提供工作证明来获得 shares，如 Puddinpop 方式一样，当区块生成时马上进行支付。但是不像之前的方式，一个区块的 shares，会被再次利用来生成下一个区块。为了区分参与矿工的交易传输费用，只有当矿工的余额超过 1BTC 时才进行支付。如果没有达到 1BTC，那么将在下一个区块生成时进行累计。如果矿工在一周内没有提供一个 share，那么矿池会将剩下的余额进行支付，不管余额是多少。
- Triplemining 方式：该方式是将一些中等大小矿池的计算力合并起来，然后将获得奖励的 1% 按照各个矿池计算力的比例分发给矿池运营者。
- P2Pool 方式：P2Pool 的挖矿节点工作在类似于比特币区块链的一种 shares 链上。由于没有中心，所以也不会受到 DoS 攻击。和其他现有的矿池技术都不一样，在该方式下，每个节点工作的区块，都包括支付给前期 shares 的所有者及该节点自己的比特币。99% 的奖励（50BTC + 交易费用）会平均分配给矿工，另外 0.5% 会奖励给生成区块的人。
- Puddinpop 方式：一种使用“元哈希”技术的方式，使用特定的 Puddinpop 挖矿软件，现在已经没有矿池使用这种方式了。

目前使用较多的方式为 Pay-Per-Share（PPS），矿工使用起来也比较方便。

但从去中心化的角度来说，还是推荐 P2Pool，在避免 DoS 攻击的同时，也能防止个别矿池拥有超大的计算力而对比特币网络造成威胁。不过 P2Pool 的使用方式较 PPS 更为复杂。

1.4　脚本系统

比特币在交易中使用脚本系统，与 FORTH（一种编译语言）一样，脚本是简单的、基于堆栈的，并且是从左向右处理的，它特意设计成非图灵完整的形式，没有 LOOP 语句。

一个脚本本质上是众多指令的列表，这些指令记录在每个交易中，若交易的接收者想花掉发送给他的比特币，那么这些指令就是描述接收者是如何获得这些比特币的。一个典型的发送比特币到目标地址 D 的脚本，要求接收者提供以下两个条件，才能花掉发给他的比特币：

1）一个公钥，当进行散列生成比特币地址时，生成的地址是嵌入在脚本中的目标地址 D。

2）一个签名，用于证明接收者保存了与上述公钥相对应的私钥。

脚本可以灵活地改变花掉比特币的条件，举个例子，脚本系统可能会同时要求两个私钥或几个私钥，或者无需任何私钥等。

如果联合脚本中未导致失败并且堆栈顶元素为真（非零），则表明交易有效。原先发送币

的一方，控制脚本运行，以便比特币在下一个交易中使用。想花掉币的另一方必须把以前记录的运行为真的脚本，放到输入区。

堆栈保存着字节向量，当用作数字时，字节向量被解释成小尾序的变长整数，最重要的位用于决定整数的正负号。比如，0x81 代表 -1，0x80 则是 0 的另外一种表示方式（称之为负 0）。正 0 用一个 NULL 长度向量表示。字节向量可以解析为布尔值，这里 False 表示为 0，True 表示为非 0。

1.4.1 脚本特点

表 1-4 至表 1-12 是脚本的所有关键字列表（命令 / 函数），一些更复杂的操作码已被禁用，不再考虑，因为钱包客户在这些操作码的程序实现上可能有 Bug，如果某个交易使用了这些操作码，那么其将会使比特币块链产生分叉。我们提到脚本的时候，通常省略了这些把数字压入堆栈的关键字。

表 1-4 脚本常见关键字字段

关键字	操作码	十六进制	输 入	输 出	描 述
OP_0/OP_FALSE	0	0x00	无	空	一个字节空串被推到堆栈中（并非 no-op 操作，这里有一个元素被压入堆栈）
N/A	1-75	0x01-0x4b	（特殊）	数据	下一个操作码字节是要被压入堆栈的数据
OP_PUSHDATA1	76	0x4c	（特殊）	数据	下一个字节是要被压入堆栈的数据的长度
OP_PUSHDATA2	77	0x4d	（特殊）	数据	下两个字节是要被压入堆栈的数据的长度
OP_PUSHDATA4	78	0x4e	（特殊）	数据	下四个字节是要被压入堆栈的数据的长度
OP_1NEGATE	79	0x4f	无	-1	数字 -1 被压入堆栈
OP_1, OP_TRUE	81	0x51	无	1	数字 1 被压入堆栈
OP_2-OP_16	82-96	0x52-0x60	无	2-16	与关键名相对应的数字被压入堆栈

表 1-5 所示的是脚本的流程控制。

表 1-5 脚本的流程控制

关键字	操作码	十六进制	输 入	输 出	描 述
OP_NOP	97	0x61	无	无	无任何操作
OP_IF	99	0x63	<expression> if [statements] [else [statements]]* endif		如果栈顶元素值不为 0，则语句将被执行，栈顶元素值将被删除
OP_NOTIF	100	0x64	<expression> if [statements] [else [statements]]* endif		如果栈顶元素值为 0，则语句将被执行，栈顶元素值将被删除
OP_ELSE	103	0x67	<expression> if [statements] [else [statements]]* endif		如果前述的 OP_IF 或 OP_NOTIF 或 OP_ELSE 未被执行，那么这些语句就会被执行；如果前述的 OP_IF 或 OP_NOTIF 或 OP_ELSE 被执行，那么这些语句就不会被执行
OP_ENDIF	104	0x68	<expression> if [statements] [else [statements]]* endif		结束 if/else 语言块

（续）

关键字	操作码	十六进制	输　入	输　出	描　述
OP_VERIFY	105	0x69	true/false	无 /false	如果栈顶元素值非真，则标记交易无效。true 会被删除，false 不会被删除
OP_RETURN	106	0x6a	无	无	标记交易无效

表 1-6　脚本的堆栈处理

关键字	操作码	十六进制	输　入	输　出	描　述
OP_TOALTSTACK	107	0x6b	x1	(alt) x1	把输入压入辅堆栈的顶部，从主堆栈删除
OP_FROMALTSTACK	108	0x6c	(alt) x1	x1	把输入压入主堆栈的顶部，从辅堆栈删除
OP_IFDUP	115	0x73	x	x/x x	如果栈顶元素值不为 0，则复制该元素值
OP_DEPTH	116	0x74	无	< 堆栈大小 >	把堆栈元素个数压入堆栈
OP_DROP	117	0x75	x	无	删除栈顶元素
OP_DUP	118	0x76	x	x x	复制栈顶元素
OP_NIP	119	0x77	x1 x2	x2	删除栈顶的下一个元素
OP_OVER	120	0x78	x1 x2	x1 x2 x1	复制栈顶的下一个元素到栈顶
OP_PICK	121	0x79	xn ... x2 x1 x0 <n>	xn ... x2 x1 x0 xn	把堆栈的第 n 个元素复制到栈顶
OP_ROLL	122	0x7a	xn ... x2 x1 x0 <n>	... x2 x1 x0 xn	把堆栈的第 n 个元素移动到栈顶
OP_ROT	123	0x7b	x1 x2 x3	x2 x3 x1	栈顶的三个元素向左翻转
OP_SWAP	124	0x7c	x1 x2	x2 x1	栈顶的两个元素交换
OP_TUCK	125	0x7d	x1 x2	x2 x1 x2	把栈顶元素复制并插入到栈顶下一个元素之前
OP_2DROP	109	0x6d	x1 x2	无	删除栈顶两个元素
OP_2DUP	110	0x6e	x1 x2	x1 x2 x1 x2	复制栈顶两个元素
OP_3DUP	111	0x6f	x1 x2 x3	x1 x2 x3 x1 x2 x3	复制栈顶三个元素
OP_2OVER	112	0x70	x1 x2 x3 x4	x1 x2 x3 x4 x1 x2	把栈底的两个元素复制到栈顶
OP_2ROT	113	0x71	x1 x2 x3 x4 x5 x6	x3 x4 x5 x6 x1 x2	把第五和第六个元素移动到栈顶
OP_2SWAP	114	0x72	x1 x2 x3 x4	x3 x4 x1 x2	以一对元素为单位，交换栈顶的两对元素的位置

表 1-7 展示的是字符串操作码，若有在交易中出现了已禁用的操作码，则必须终止和失败返回。

表 1-7　字符串操作码

关键字	操作码	十六进制	输　入	输　出	描　述
OP_CAT	126	0x7e	x1 x2	out	连接两个字符串，已禁用
OP_SUBSTR	127	0x7f	in begin size	out	返回字符串的一部分，已禁用

（续）

关键字	操作码	十六进制	输 入	输 出	描 述
OP_LEFT	128	0x80	in size	out	在一个字符串中保留左边指定长度的子串，已禁用
OP_RIGHT	129	0x81	in size	out	在一个字符串中保留右边指定长度的子串，已禁用
OP_SIZE	130	0x82	in	in size	把栈顶元素的字符串长度压入堆栈（无须弹出元素）

表 1-8 展示的是位操作码，若有在交易中出现了已禁用的操作码，则必须终止和失败返回。

表 1-8 位操作码

关键字	操作码	十六进制	输 入	输 出	描 述
OP_INVERT	131	0x83	in	out	所有输入的位取反，已禁用
OP_AND	132	0x84	x1 x2	out	对输入的所有位进行布尔与运算，已禁用
OP_OR	133	0x85	x1 x2	out	对输入的每一位进行布尔或运算，已禁用
OP_XOR	134	0x86	x1 x2	out	对输入的每一位进行布尔异或运算，已禁用
OP_EQUAL	135	0x87	x1 x2	true/false	如果输入的两个数相等，则返回 1，否则返回 0
OP_EQUALVERIFY	136	0x88	x1 x2	true/false	与 OP_EQUAL 一样，之后运行 OP_VERIFY

注意 算术逻辑的输入仅限于有符号 32 位长整数，但输出有可能会溢出。如果任何命令的输入值其长度超过 4 字节，那么脚本必须中止和失败返回。如果在交易中出现了标记为已禁用的操作码，也必须终止和失败返回。

表 1-9 算术逻辑操作码（若在交易中出现已禁用的操作码，则必须终止和失败返回）

关键字	操作码	十六进制	输入	输出	描述
OP_1ADD	139	0x8b	in	out	输入值加 1
OP_1SUB	140	0x8c	in	out	输入值减 1
OP_2MUL	141	0x8d	in	out	输入值乘 2，已禁用
OP_2DIV	142	0x8e	in	out	输入值除 2，已禁用
OP_NEGATE	143	0x8f	in	out	输入值符号取反
OP_ABS	144	0x90	in	out	输入值符号取正
OP_NOT	145	0x91	in	out	如果输入值为 0 或 1，则输出 1 或 0；否则输出 0
OP_0NOTEQUAL	146	0x92	in	out	输入值为 0 输出 0；否则输出 1
OP_ADD	147	0x93	a b	out	输出 a + b
OP_SUB	148	0x94	a b	out	输出 a – b
OP_MUL	149	0x95	a b	out	输出 a × b，已禁用
OP_DIV	150	0x96	a b	out	输出 a/b，已禁用
OP_MOD	151	0x97	a b	out	输出 a/b 的余数，已禁用
OP_LSHIFT	152	0x98	a b	out	把 a 向左移动 b 位，保留符号，已禁用
OP_RSHIFT	153	0x99	a b	out	把 a 向右移动 b 位，保留符号，已禁用

（续）

关键字	操作码	十六进制	输入	输出	描述
OP_BOOLAND	154	0x9a	a b	out	如果 a 和 b 都不为 0，则输出 1，否则输出 0
OP_BOOLOR	155	0x9b	a b	out	如果 a 或 b 不为 0，则输出 1，否则输出 0
OP_NUMEQUAL	156	0x9c	a b	out	如果 a = b 则输出 1，否则输出 0
OP_NUMEQUALVERIFY	157	0x9d	a b	out	与 OP_NUMEQUAL 一样，之后要运行 OP_VERIFY
OP_NUMNOTEQUAL	158	0x9e	a b	out	如果 a! = b 则输出 1，否则输出 0
OP_LESSTHAN	159	0x9f	a b	out	如果 a < b 则输出 1，否则输出 0
OP_GREATERTHAN	160	0xa0	a b	out	如果 a > b 则输出 1，否则输出 0
OP_LESSTHANOREQUAL	161	0xa1	a b	out	如果 a <= b 则输出 1，否则输出 0
OP_GREATERTHANOR-EQUAL	162	0xa2	a b	out	如果 a >= b 则输出 1，否则输出 0
OP_MIN	163	0xa3	a b	out	输出 a、b 中的最小值
OP_MAX	164	0xa4	a b	out	输出 a、b 中的最大值
OP_WITHIN	165	0xa5	x min max	out	如果 x 在 min 和 max 之间，则输出 1，否则输出 0

表 1-10　脚本的加密码

关键字	操作码	十六进制	输　入	输　出	描　述
OP_RIPEMD160	166	0xa6	in	Hash	输入用 RIPEMD-160 算法散列
OP_SHA1	167	0xa7	in	Hash	输入用 SHA-1 算法散列
OP_SHA256	168	0xa8	in	Hash	输入用 SHA-256 算法散列
OP_HASH160	169	0xa9	in	Hash	输入被散列两次，先用 SHA-256，再用 RIPEMD-160
OP_HASH256	170	0xaa	in	Hash	输入用 SHA-256 算法散列两次
OP_CODESEPA-RATOR	171	0xab	无	无	所有签名检查只需要匹配最近一次执行的 OP_CODESEPARATOR 操作数据的签名即可
OP_CHECKSIG	172	0xac	sig pubkey	true/false	全部交易的输出、输入和脚本都被散列，OP_CHECKSIG 使用的签名必须是该 Hash 值和公钥的有效签名，如果是真则返回 1，否则返回 0
OP_CHECKSIG-VERIFY	173	0xad	sig pubkey	true/false	与 OP_CHECKSIG 一样，但之后执行 OP_VERIFY
OP_CHECKMUL-TISIG	174	0xae	x sig1 sig2 ... <number of signatures> pub1 pub2 <number of public keys>	true/false	对于每个签名和公钥对，OP_CHECKSIG 都会被执行，如果公钥数量比签名多，那么一些公钥 / 签名对会失败。所有的签名要与公钥匹配。如果所有签名都有效，则输出 1，否则返回 0。因为存在 BUG，因此一个未使用的外部值会从堆栈中删除
OP_CHECKMU-LTISIGVERIFY	175	0xaf	x sig1 sig2 ... <number of signatures> pub1 pub2 ... <number of public keys>	true/false	与 OP_CHECKMULTISIG，一样，但是之后运行 OP_VERIFY

表 1-11 脚本中的伪关键字（这些关键字仅供内部使用，辅助进行交易匹配）

关键字	操作码	十六进制	描　述
OP_PUBKEYHASH	253	0xfd	表示公钥用 OP_HASH160 操作码散列
OP_PUBKEY	254	0xfe	表示与 OP_CHECKSIG 兼容的一个公钥
OP_INVALIDOPCODE	255	0xff	匹配任何未指定的操作码

表 1-12 保留关键字

关键字	操作码	十六进制	描　述
OP_RESERVED	80	0x50	交易无效，除非发生在未执行的 OP_IF 分支
OP_VER	98	0x62	交易无效，除非发生在未执行的 OP_IF 分支
OP_VERIF	101	0x65	交易无效，即使发生在未执行的 OP_IF 分支
OP_VERNOTIF	102	0x66	交易无效，即使发生在未执行的 OP_IF 分支
OP_RESERVED1	137	0x89	交易无效，除非发生在未执行的 OP_IF 分支
OP_RESERVED2	138	0x8a	交易无效，除非发生在未执行的 OP_IF 分支
OP_NOP1-OP_NOP10	176-185	0xb0-0xb9	这些关键字被忽略

1.4.2 脚本运行过程

在这里，我们先讨论单输入单输出的比特币交易，因为这样描述起来更为方便，而且不会影响对脚本的理解，以下面的一个交易 Hash 值为例：

9c50cee8d50e273100987bb12ec46208cb04a1d5b68c9bea84fd4a04854b5eb1

这是一个单输入单输出交易，下面来看下我们要关注的数据。

Hash：

9c50cee8d50e273100987bb12ec46208cb04a1d5b68c9bea84fd4a04854b5eb1

对于输入交易，需要关注如下值。

前导输入的 Hash：

437b95ae15f87c7a8ab4f51db5d3c877b972ef92f26fbc6d3c4663d1bc750149

输入脚本 scriptSig：

```
3045022100efe12e2584bbd346bccfe67fd50a54191e4f45f945e3853658284358d9c062ad02200
121e00b6297c0874650d00b786971f5b4601e32b3f81afa9f9f8108e93c752201038b29d4fbbd12619d
45c84c83cb4330337ab1b1a3737250f29cec679d7551148a
```

对于输出交易，需要关注如下值。

转账金额：0.05010000 btc

输出脚本 scriptPubKey：

```
OP_DUP OP_HASH160 be10f0a78f5ac63e8746f7f2e62a5663eed05788 OP_EQUALVERIFY OP_CHECKSIG
```

假设 Alice 是转账发送者，Bob 是接受者。那么输入交易表明了 Alice 要动用的比特币的来源，交易输出表明了 Alice 要转账的数额和转账对象——Bob。那么，有读者可能会问，数

据中的输入脚本和输出脚本是不是就是题和解？答对了一半！

在 Bitcoin Wiki 中提到：原先发送币的一方，控制脚本运行，以便比特币在下一个交易中使用。想花掉币的另一方必须把以前记录的运行为真的脚本，放到输入区。

换句话说，在一个交易中，输出脚本是数学题，输入脚本是题解，但不是这道数学题的题解。我开始看 Wiki 的时候，在这里遇到了一些障碍，没法理解输入脚本和输出脚本的联系。但是在考虑到交易间的关系之后，就明白了。

假设有这么一系列交易，如图 1-7 所示。

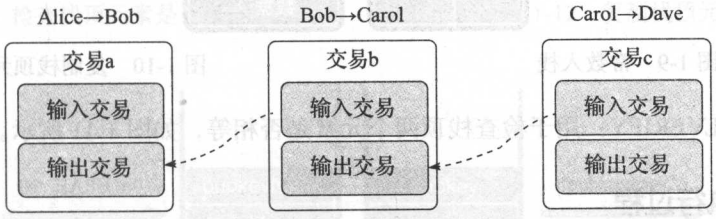


图 1-7　三对交易示范图

那么，这一系列交易具有如下特征。

- 三个交易都是单输入单输出交易。
- 每个输入交易输出交易中，都包含对应的脚本。
- 交易 a 为 Alice 转账给 Bob；交易 b 为 Bob 转账给 Carol；交易 c 为 Carol 转账给 Dave。
- 当前交易的输入都引用前一个交易的输出，如交易 b 的输入就是引用交易 a 的输出。

按照之前的说法，交易 a 中的输出脚本就是 Alice 为 Bob 出的数学题。那么，Bob 想要引用交易 a 输出交易的比特币，就要解开这道数学题。题解是在交易 b 的输入脚本里给出的！ Bob 解开了这道题，获得了奖金，然后在交易 b 中为 Carol 出一道数学题，等待 Carol 来解……

所以说，在图 1-8 中相同颜色的输出和输入才是一对题和解。

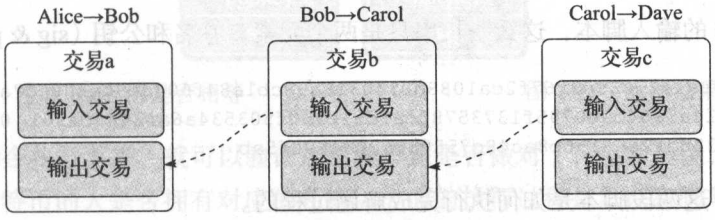


图 1-8　输入输出一一对应图

1.4.3　脚本操作码解读

要理解比特币脚本，先要了解堆栈，这是一个后进先出（Last In First Out）的容器，脚本系统对数据的操作都是通过它完成的。比特币脚本系统中有两个堆栈：主堆栈和副堆栈，一

般来说主要使用主堆栈。下面就来列举几个简单的例子，看下指令是如何对堆栈进行操作的。

常数入栈：指把一段常数压入到堆栈中，这个常数成为栈顶元素，如图 1-9 所示。

OP_DUP：复制栈顶元素，如图 1-10 所示。

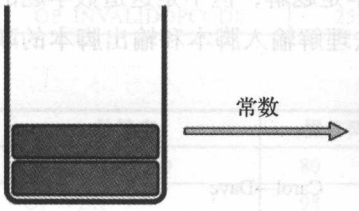


图 1-9 常数入栈

图 1-10 复制栈顶元素

OP_EQUALVERIFY：用于检查栈顶两个元素是否相等，如图 1-11 所示。

1.4.4 脚本执行过程

Alice 在转账给 Bob 的时候，输出交易中给出了 Bob 的钱包地址（等价于公钥 Hash），当 Bob 想要转账给 Carol 的时候，他要证明自己拥有与这个钱包地址对应的私钥，所以在输入交易中给出了自己的公钥及使用私钥对交易的签名。下面来看个实例。

交易 a：

```
9c50cee8d50e273100987bb12ec46208cb04a1d5b68c9bea84fd4a04854b5eb1
```

交易 b：

```
62fadb313b74854a818de4b4c0dc2e2049282b28ec88091a9497321203fb016e
```

交易 b 中有一个输入交易引用了交易 a 的输出交易，它们的脚本是一对题与解。

题：交易 a 的输出脚本，若干个脚本指令和转账接收方的公钥 Hash。

```
OP_DUPOP_HASH160be10f0a78f5ac63e8746f7f2e62a5663eed05788OP_EQUALVERIFY OP_CHECKSIG
```

解：交易 b 的输入脚本，这么一长串只是两个元素，签名和公钥（sig & pubkey）。

```
3046022100ba1427639c9f67f2ca1088d0140318a98cb1e84f604dc90ae00ed7a5f9c61cab02210
094233d018f2f014a5864c9e0795f13735780cafd51b950f503534a6af246aca301 03a63ab88e75116
b313c6de384496328df2656156b8ac48c75505cd20a4890f5ab
```

下面来看下这两段脚本是如何执行完成解题过程的。

首先执行的是输入脚本。因为脚本是从左向右执行的，那么先入栈的是签名，随后是公钥。接着，执行的是输出脚本。从左向右执行，第一个指令是 OP_DUP——复制栈顶元素（如图 1-12 所示）。

OP_HASH160 用于计算栈顶元素 Hash，得到 pubkeyhash，如图 1-13 所示。

然后将输出脚本中的公钥 Hash 入栈，为了与前面计算中所得到的 Hash 区别开来，这里

称它为 pubkeyhash'，如图 1-14 所示。

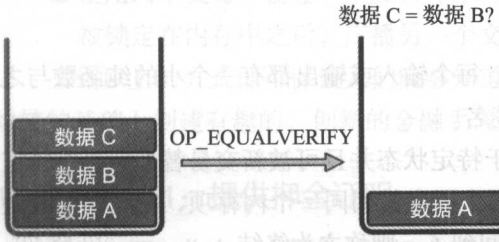


图 1-11　检查栈顶元素是否相等

图 1-12　复制栈顶元素

图 1-13　栈顶元素 Hash160

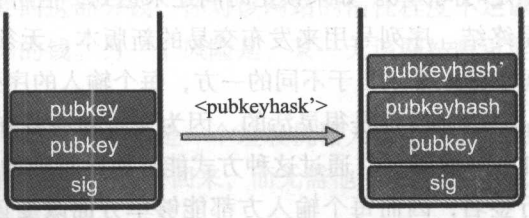


图 1-14　公钥 Hash 入栈

OP_EQUALVERIFY 则会检查栈顶前两个元素是否相等，如果相等则继续执行，否则中断执行，返回失败，如图 1-15 所示。

OP_CHECKSIG 使用栈顶前两个元素执行签名校验操作，如果相等，则返回成功，否则返回失败，如图 1-16 所示。

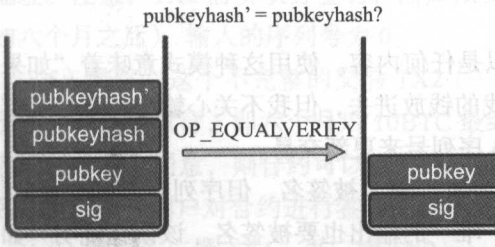


图 1-15　检查 Hash 值是否相等

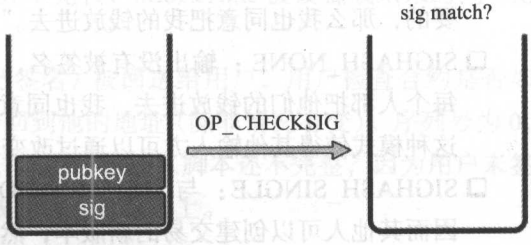


图 1-16　返回结果

这样一串指令执行下来，就可以验证这道数学题是否做对了，也就是说，验明了想要花费钱包地址中比特币的人是否拥有对应的私钥。上面的执行过程是可以在脚本模拟器中进行的，并且能够看到每一步执行的状态。

1.5　合约应用案例

1.4 节的脚本系统，详细说明了脚本的运行原理，本节将描述在实际应用场景中如何使

用脚本系统构建合约应用。

1.5.1 合约应用原理

每个比特币交易都有一个或多个输入和输出，每个输入或输出都有一个小的纯函数与之相关联，称为脚本，脚本可包含简化形式交易的签名。

每个交易都有一个锁定时间，使得该交易处于特定状态并且可被新交易替换，直至锁定时间来临。预定时间可以是块索引或时间戳（这两个因素使用同一个内存项，小于5亿是块索引，大于5亿是时间戳）。当一个交易的锁定时间到了，则称之为终结。

每个交易的输入都有一个序列号，对于正常发送币的交易，该序列号为UNIT_MAX，锁定时间为0。如果锁定时间还未达到，但所有的序列号为UNIT_MAX，则该交易也被认为是终结。序列号用来发布交易的新版本，无须验证其他输入的签名。例如：在一个交易中，每个输入都来自于不同的一方，每个输入的序列号都是0，这些序列号可以独立增加。

签名验证是很灵活的，因为交易的签名方式可以通过SIGHASH符号来控制，该符号附加在签名后面。通过这种方式能够构建特殊的合约，交易的每个输入方只对交易的一部分进行签名，因而每个输入方都能够单方面改变该交易的一部分内容，而无需其他输入方的参与。SIGHASH符号分为两部分，一种模式和一个ANYONECANPAY指示器。签名模式包含如下几种模式。

- SIGHASH_ALL：这是默认模式。它指示一个交易除输入脚本之外，所有部分都被签名。对输入脚本进行签名显然是不可能的，那样将无法构建一个交易，所以脚本总是不被签名。尽管如此，需要注意的是，输入的其他属性如输出、序列号等都会被签名。直观地说，它的意思是“如果每个人都把他们的钱放进去，并且输出的正是我想要的，那么我也同意把我的钱放进去。”
- SIGHASH_NONE：输出没有被签名，可以是任何内容。使用这种模式意味着“如果每个人都把他们的钱放进去，我也同意把我的钱放进去，但我不关心输出的是什么。”这种模式使得其他输入方可以通过改变输入序列号来更新交易。
- SIGHASH_SINGLE：与SIGHASH_NONE一样，输入被签名，但序列号没有被签名。因而其他人可以创建交易的新版本，然而，唯一的输出也要被签名。该模式说明“如果输出的正是我想要的，那么我同意把钱放进去，但我不关心其他人的输入。”

ANYONECANPAY指示器可以与以上三种模式联合使用，当设置了ANYONECANPAY时，仅仅是该输入被签名，其他输入可以是任意内容。

脚本可以包括CHECKMULTISIG操作码，该操作码提供了n-of-m的签名验证，即：你可以提供多个公钥m，定义必须出现的有效签名个数n，签名个数n可以小于公钥数量m。如果我们设置以下脚本，则一个输出需要两个签名：

```
2<pubkey1><pubkey2>2CHECKMULTISIGVERIFY
```

有如下两种通用的方式可用来安全地创建合约。

- 在 P2P 网络之外传递部分完成或无效的交易。
- 使用两个交易：创建一个交易（合约交易），先签名但不会马上广播，在达成合约并且被锁定在内存中之后，广播另一个交易（支付交易），最后再广播合约。

采用以上的方式即可保证人们能够知道他们达成的合约内容。这些特性可以让我们在区块链的基础上创建有趣的、创新的金融手段。

1.5.2　示例 1：提供押金证明

想象一下，你在一个网站（论坛或 WIKI）上注册了一个账号，现在你希望在网站运营者处建立你的信用，但是你没有以前的名誉来支撑你的信用。一个解决方案就是向网站付点钱购买信用，但是如果你关闭了账号，可能会想要回这部分钱。你对该网站的信任程度不足以让你将钱存到该网站，因为你担心网站会花掉你的钱。另一个风险是，某一天该网站有可能会消失。

建立信用度的目的是你做出某种奉献，让网站知道你不是一个垃圾机器人。但是你不想让网站花掉你的钱。如果网站运营者消失了，你最终想把钱要回来，而无需他们的任何许可。

对于该问题，可以通过合约来解决，具体步骤如下。

1）用户和网站相互发送各自新生成的公钥。

2）用户创建交易 TX1（支付交易），该交易支出 10 个 BTC 到网站地址，用户创建了 TX1 但不广播。

3）用户把 TX1 交易的 Hash 值发送给网站。

4）网站使用 TX1 的 Hash 值创建交易 TX2（合约），TX2 花掉 TX1 的钱并且支付到用户地址。注意，TX2 需要双方签名，因而该交易并不完整，nLockTime 被设置成未来时间（比如六个月之后），输入的序列号为 0。

5）最终，这个不完整的交易 TX2（一半已签名）被回送给用户，用户检查合约是否如预期的一样在执行，即六个月后 10BTC 最终会回到他的地址（除非情况有变）。序列号为 0，表示如果双方同意，则合约可以被修订。现在，该交易的输入脚本还不完整，因为用户未签名，所以要等用户对合约进行签名并且把签名放到合适的位置上。

6）用户先广播 TX1，再广播 TX2。

在这个阶段，用户和网站都不能单独得到 10BTC。六个月之后，合约完成，即使网站消失了，用户也能得到币。

如果用户想要提早关闭账号，又该怎么处理呢？网站创建新版的 TX2，nLockTime 设为 0，并且输入的序列号设为 UINT __ MAX，重新签名，把该交易发回用户，用户签名后广播该交易，就能提早结束合约并且释放 10BTC。

如果六个月快到了，而用户还想保留他的账号，又该怎么办呢？类似的事情发生后，合约会使用新的 nLockTime，序列号比以前的序列号大 1，双方重新签名，广播 2^{32} 次。当然，无论发生什么，双方都必须同意，才能真正改变合约。

显然，如果该用户被证明是存在恶意行为的（例如：垃圾邮件发送者），那么网站不会同意提早结束合约。如果用户有太多的滥用行为，则网站可以要求增加存款数量，或者要求延长合约时间。

1.5.3 示例2：担保和争端调解

一个买家想和他不认识或不信任的某人进行交易，一般情况下若交易能够正常进行时，买家不想任何第三方参与。但是当交易出现问题时，他想有一个第三方——也许是一个专业的争端调解服务来决定谁能拿到钱。


注意 这个概念同时适用于买家和卖家。例如，调解员可向商家要求邮资证明，以判断是否发货。

换句话说，某人想锁定某些币时，这些币要在第三方同意的情况下，才能被花掉。

该示例的实现步骤具体如下。

1）和商家一起引入一个调解员（如：ClearCoin）。

2）得到商家的公钥K1，得到调解员的公钥K2，创建自己的公钥K3。

3）把K2发给商家，商家生成一个随机数挑战调解员，调解员用K2的私钥签名，用来证明K2确实属于调解员。

4）创建一个交易TX1，使用如下输出脚本并且广播该交易。

```
2<K1><K2><K3>3CHECKMULTISIGVERIFY
```

现在这些币被锁定了，如果要解锁这些币，需要使用以下几种方式。

- 客户和商家同意（无论是成功的交易，还是在没有调解的情况下商家同意回退给客户）。
- 客户和调解者同意（失败的交易，调解者认同客户，客户得到退款）。
- 调解者和商家同意（商品已经发送，尽管有争议，商家还是得到币）。

输入签名时，内容被设为相关联的输出。这样，为了从这个交易中得到币，客户要创建包含两个签名位的脚本，自己签一个，再把未完成的交易发给商家或调解员，请求第二个签名。

1.5.4 示例3：保证合约

保证合约是建造公众商品时的集资办法，公众商品是指一旦建成，任何人都可以免费享受到好处的商品。标准的例子是灯塔，所有人都认同应该建造一个，但是对于航海者个人来说灯塔太贵了，而且灯塔不只是他一个人用得着，同时也会方便其他的航海者。

一个解决方案就是向所有人集资，只有当筹集的资金超过所需的建造成本时，每个人才真正付钱；如果集资款不足，则谁都不用付钱。

在保证合约集资方面，包括频繁的、小额的、经常自动进行的集资，例如互联网电台的资金和网页翻译等，比特币要优于传统的支付方式。假设有一个浏览器的插件可供你发送一点币，它能检测当前页面的语言并且广播一个集资请求，用于把该页面翻译成你的语言。如果使用该插件的许多用户同时查看该页面（例如：该页面从高流量的网站链接过来），那么足够的集资请求就能广播出去，到达一定的金额时，可自动付钱给一个高质量的翻译公司，当翻译完成后该页面将自动在你的浏览器中加载。

我们能以比特币的方式建立以下模型，具体步骤如下。

1）主办方创建新的捐赠地址，宣布如果筹集资金超过 1000BTC，则将建造该商品，任何人都可以捐赠。

2）捐赠者创建一个新交易，把一定数量的钱打到集资地址上，但是他们并不广播该交易。该交易与常规的交易相似，但有三个不同点：首先，不能做任何改变，如果你没有正确的输出金额 1000BTC，那么你必须先创建一个；第二，输入脚本要以 SIGHASH_ALL|SIGHASH_ANYONECANPAY 的模式签名；最后，输出值是 1000BTC，注意，这不是一个有效的交易，因为输出值比输入值大得多。

3）把交易上传到主办方的服务器上，他们把交易保存到磁盘上，随时更新捐赠的币数量。

4）一旦服务器获得了足够的币，它将把所有捐赠者上传的独立交易合并成一个新的交易，该交易只有一个输出，仅仅是把钱付到捐赠地址，该输出与每个捐赠者的交易的输出部分相同，而输入部分则是所有捐赠者输入的集合。

5）广播完整的交易，发送捐赠的币到捐赠地址中。

这样的场景依靠了协议的几个方面，首先使用了 SIGHASH 符号，SIGHASH_ALL 是默认模式，意味着要签名所有交易的内容，除了输入脚本。SIGHASH_ANYONECANPAY 是附加的指示器，意味着签名仅覆盖自己的输入部分，而不会覆盖其他人的输入，这样一来，其他人的输入可以留空。使用这些符号，我们能创建这样一个签名，即使在添加进其他输入之后，该签名依旧是有效的。但如果输出内容或其他的交易部分被改变了，那么该签名就会无效了。第二，输入值小于输出值的交易是无效的（原因很明显），这也就意味着捐赠者把发送币的交易发送给主办方是安全的，因为主办方不可能得到这些捐赠币，除非再加上其他的输入值等于或超过输出值。

不使用 SIGHASH_ANYONECANPAY 指示器也可以创建保证合约。不需捐赠者创建交易的集资方式有两个步骤，一旦达到集资的总金额，主办方就会创建包含所有捐赠者输入的交易，然后依次在捐赠者中传递，每个捐赠者都对该交易进行签名。但是，先使用 SIGHASH_ANYONECANPAY 指示器，然后合并交易，这样可能会更方便一些。

保证合约可以保证下一个块的资金网络安全，通过这种方式，即使一个块中的交易数量比较少，挖矿也能挣钱（因为保证合约的交易一般占用空间较大，因而需要付出更多的网络转账费）。

Tabarrok 在他的论文“The private provision of public goods via dominant assurance contracts”

中详尽地描述了保证合约的概念，在一个通用的保证合约中，如果合约失败了（在预定时间内集资不足），主办方会给捐赠者支付网络转账费，这种类型的合约旨在鼓励捐赠者积极参与。

1.5.5 示例4：使用外部状态

脚本被设计成纯函数，它们不能访问外部服务器，也不能导入任何会改变的外部状态，因为攻击者会利用该特性突破区块链。而且，脚本语言的功能被严格限制了。幸运的是，我们可以以其他的方式创建交易，从而与外部世界相联系。

考虑一个例子，老人想让他的孙子继承遗产，继承时间是在他死后，或者在孙子年满18岁时，无论先满足哪个条件，他的孙子都可以得到遗产。

为了解决这个问题，老人首先向他自己发送孙子要继承的资产数量，以便有一个正确的继承数量的唯一输出；接着，他创建了一个带有锁定时间的交易，该交易的意思是：在孙子18岁生日时，把币支付到孙子的地址中，老人对该交易签名，不进行广播，直接把该交易给了孙子。当过了孙子的18岁生日之后，孙子广播了这个交易并且得到他的币。孙子可以在这个时间之前广播该交易，但他不会提前得到币，有些节点会在内存池中把这种交易丢弃掉，因为锁定时间在遥远的将来。

死亡条件则很难判断，因为比特币节点不会检测主观条件，我们必须依靠预言，预言是指具有密钥对的服务器，当用户自定义的表达式被证明是真的，它就能按照要求对交易进行签名。

以下是例子，老人创建了一个交易花掉了他的币，把输出设为：

```
<hash>OP_DROP2<sonspubkey><oraclepubkey>CHECKMULTISIG
```

这是一个预言脚本，具有与众不同的形式——它把数据推到堆栈中，然后立即删除。公钥发布在预言服务器的网站上，为公众所知，Hash值设为用户自定义表达式的Hash值，该表达式以预言服务器能够理解的方式进行编写，以确认老人已经死亡。举个例子，可能是以下字符串的Hash值：

```
if(has_died('johnsmith',born_on=1950/01/02))return(10.0,1JxgRXEHBi86zYzHN2U4KMyRCg4LvwNUrp);
```

以上语言是假设的，由预言服务器定义该语言，其可能是任何一种语言。返回值是一个输出：币数量和孙子的地址。

再一次，老人创建了一个交易，没有广播而是直接给了孙子，他另外还提供了一个表达式和预言服务器的名字，Hash值被写入交易，预言服务器则能解锁表达式。算法的具体实现如下。

1）预言服务器接受评估请求，请求包括了用户提供的用户自定义表达式、输出脚本和部分完成的交易，交易中除了scriptSig签名之外，其他部分都已经完成，签名仅包括了孙子的签名，还不足以解锁输出。

2）预言服务器检查表达式，得到其 Hash 值，并且与交易中的 Hash 值对照，如果两者不一致，则返回错误。

3）预言服务器评估表达式，如果表达式的结果不是交易输出的目标地址，则返回错误。

4）预言服务器签名交易，返回签名给用户。

注意　对一个比特币交易进行签名时，输入脚本被设为相关联的输出脚本。原因是当 OP_CHECKSIG 操作起作用时，包含该操作码的脚本被放到输入中，脚本并不包含签名本身。预言服务器并不知道完整的输出，也没有必要知道，因为它知道输出脚本、自己的公钥和表达式的 Hash 值，这就足以使它检查输出脚本并且完成交易。

5）用户接受新签名，插入到交易的 scriptSig 项中，广播交易。

当且仅当预言服务器认为老人死了，孙子才能广播两个交易（合约和申报），并且得到币。

预言服务器可以评估任何事情，然而区块链中的输出脚本的形式总是一样的，可考虑以下的可能性。

- 给定一个日期，假定 mtgox 比特币的美元价格在 12.5 ~ 13.5 之间：

```
today()==2011/09/25&&exchange_rate(mtgoxUSD)>=12.5&&exchange_rate(mtgox-
USD)<=13.5
```

- 打赌我会做一些我从来不会做的事情（比如，获得奥林匹克金牌）：

```
google_results_count(site:www.google.com/hostednews'MikeHearn'Olympicgoldmedal)>0
```

以上函数通过 Google 来搜索 MikeHearn 这个人获得奥林匹克金牌（Olympicgoldmedal）的块数

- 欧洲电视台的歌唱比赛，选择两个优胜者之一进行打赌：

```
if(eurovision_winner()=='Azerbaijan')
  return 1Lj9udBVDwptFffGSJSC2sohCfudQgSTPD;
else
  return 1JxgRXEHBi86zYzHN2U4KMyRCg4LvwNUrp;
```

这些条件可使预言服务器的签名变得任意复杂，但是区块链仅需要包含一个 Hash 值即可。

1. 信任最小化：挑战

有许多方法可以降低对预言服务器的信任程度。

回到我们的第一个例子，预言服务器还没有看到孙子想解锁的交易，因为该交易从没被广播过。这样，它不会支持孙子赎回币，因为它不知道该交易是否存在。我们能做到，并且也应该做到，以自动方式定期挑战预言服务器，确保它总是按照我们想象的方式进行输出。挑战无须花费任何币，因为要签名的交易是无效的（例如：与一个并不存在的交易进行关联），预言服务器没法知道签名请求是随意的还是真实的，如何以一个尚未成真的条件来挑战预言

服务器，是一个开放的研究课题。

2. 信任最小化：多个独立预言服务器

如果需要，CHECKMULTISIG 中的签名个数 n 可以增加至能够达到 n-of-m 的预言服务器模式（即 m 个预言服务器中需要有 n 个预言服务器签名才能使交易有效）。当然，检查预言服务器是独立的而非串通的，这一点也是很重要的。

3. 信任最小化：可信的硬件

使用合适的硬件，即可进行信任计算。比如，以 INTELTXT 或 AMD 等硬件来建立一个封闭的运行环境，然后用 TPM 芯片向第三方证实其可信度。第三方可判定硬件是否处于所需状态。如果硬件失败，则需要有人介入 CPU 程序的执行过程，甚至在极端的情况下，内存总线没有数据流过（如果程序足够小，则完全可以使用缓存来运行程序）。

4. 信任最小化：亚马逊 AWS 预言服务器

最终，也许是目前最现实的方法就是使用亚马逊的网页服务，截至 2013 年 11 月，最合适的预言服务器的解决方案是“ this recipe for creating a trusted computing environment using AWS”，该方案基于“this project for doing selective SSL logging and decryption”。基本思想是：预言服务器使用亚马逊作为其信任根，并且能用亚马逊 API 证明其是值得信任的，该预言服务器记录在线银行接口的加密 SSL 会话，如果以后产生交易争端，那么会话记录将被解密，争端就可以得到解决。

1.5.6 示例 5：跨链交易

比特币技术可以用来创建多个独立的货币，与比特币实现理念相同的山寨币，可以在有限信任的条件下与比特币进行自由交易。域名币（Namecoin）就是一个例子，它与比特币的运作规则有些不同，它可以在域名空间中租用域名。

举个例子，想象一个财团发行了欧元币（EURCoins），即一种以财团的银行存款 1:1 支持的加密货币。这样的货币与比特币存在不同的交易集：更中心化，但没有外汇风险。人们可能希望在比特币与欧元币之间来回交易，为了实现这个想法，可以使用 TierNolan 提出的协议。实现步骤具体如下。

1）A 产生一些随机数据 X（秘密）。

2）A 产生 TX1 交易（支付）包含了带跨链交易脚本的输出。它允许币以 A 和 B 共同签名的方式释放，也可以以私密 X 和 B 签名的方式释放，该交易未广播，块链的释放脚本包含了私密的 Hash 值，并非真正的私密 X 本身。

3）A 产生 TX2（合约），花掉 TX1 并且输出到 A 的地址，该交易有个未来的锁定时间，输入的序列号为 0，因而可以被替换。A 签名 TX2 并且发送给 B，B 给 TX2 签名后发送回 A。

4）A 广播 TX1 和 TX2，B 可以看到币但是不能花掉它们，因为并没有输出到 B 的地址，该交易还没有终结。

5）B 在山寨币块链上执行相同的操作，B 的锁定时间应该大于 A 的锁定时间，双方的交易都待定但未完全。

6）因为 A 知道私密 X，A 能马上申报他的币，然而，A 在申报币的过程中，向 B 释放了私密 X，所以 B 可以以私密 X 和签名 B 来完成山寨币块链的交易。

自动化交易完全是以点对点的方式进行的，其能确保货币的流动性，该协议使得这种交易更为灵活。跨链交易的脚本如下所示：

```
IF
  2 <keyA> <keyB> 2 CHECKMULTISIGVERIFY
ELSE
  <keyB> CHECKSIGVERIFY SHA256<hashofsecretx> EQUALVERIFY
ENDIF
```

合约输入的脚本如下所示：

```
<sigA> <sigB> 1
```

或者：

```
<secretx> <sigB> 0
```

由合约输入脚本中的 1 或 0 来决定应该使用哪种方式。如果为 1，则跨链交易的脚本执行第一段代码：

```
2 <keyA> <keyB> 2 CHECKMULTISIGVERIFY
```

如果是 0，则跨链交易的脚本执行第二段代码：

```
<keyB> CHECKSIGVERIFY SHA256<hashofsecretx> EQUALVERIFY
```

参考“Atomic cross-chain trading”（见参考资料 [9]）能够得到更详细的说明。

欧元币是一个很自然的想法，还有其他方法也能够实现点对货币的交易（把比特币换成菲亚特汽车，反之亦然），看“Ripple currency exchange”（见参考资料 [10]）可以得到更多的信息。

Sergio Demian-Lerner 提出了 P2PTradeX 协议，一种块链交易的解决方案，该方案需要把一个块链中的确认规则有效地编码进另一个块链的确认规则中。

1.5.7　示例 6：支付证明合约

在示例 4 中，我们看到了如何基于任意程序的输出来实现条件支付，这些程序非常有用，能够实现任何常规程序所能实现的功能，比如获取网页。缺点是需要一个第三方（预言服务器），尽管我们可以用技术来降低预言服务器的信任度，但谁都不能降低到 0。

对于受限的程序、纯函数，新加密技术已经出现，这些技术能从将信任度降低到 0，无需第三方参加。这些程序不能进行任何 I/O 操作，但是在许多情况下，这种限制被证明并不重要，或者可以以其他的方式绕过，比如给程序一个签过名并且打过时间戳的文档作为输入，无需程序自己从网上下载。

若想进一步详细了解，可以阅读一下该协议的解释“Zero Knowledge Contingent Payment”（见参考资料 [11]）。

1.5.8 示例 7：特定对象的快速调整（微）支付

与传统支付系统相比，比特币交易的费用非常便宜，但是还是需要一些费用以便矿工来挖矿，以及融合到区块链上。有些情况下，你可能想要快速和便宜地调整发送到某个特定地址的货币金额，而不会导致产生广播交易的费用。

举个例子，有一个你尚不信任的互联网接入点，就像你从来没有去过的咖啡屋中的一个 Wi-Fi 热点一样。每使用 10K 字节的流量你得向咖啡屋支付 0.001BTC，而无须注册咖啡屋账号。零信任解决方案意味着可以全自动完成整个过程，因而你在月初预先转了一笔钱到你的手机钱包中，然后你的手机会自动与 AP 协商并且按需给 AP 支付费用，咖啡屋也希望任何人都能来付钱给它，而无须担心被诈骗。

为了达到这个目标，可以使用下面的协议。该协议依赖 nLockTime 与原设计不同的行为，从 2013 年开始，时间锁定的交易被认为是非标准协议，不能进入内存池，这样就不能在锁定时间过期之前广播出去。当 nLockTime 的行为恢复回中本聪的初始设计时，就需要修改以下讨论的协议了。

可定义客户端为发送币的那一方，服务端为接收币的另一方，以下协议的实施过程是从客户的角度来进行的，具体步骤如下。

1）创建公钥 K1，向服务端请求公钥 K2。

2）创建、签名，但不广播交易 T1，支付 10BTC 到输出，需要服务端和你自己的公钥，使用 OP_CHECKMULTISIG 是一个好办法。

3）创建退款交易 T2，与 T1 的输出相关联，发送所有币回到你的地址。该交易有一个锁定时间，例如几小时后，不签名并且把该交易发送给服务端，常规来说，输出脚本是“2 K1 K2 2 CHECKMULTISIG”。

4）服务端用 K2 签名 T2，返回给客户，注意，服务端目前还未看到 T1，仅仅看到了 T1 的 Hash 值（该 Hash 值在未签名的 T2 中）。

5）客户验证服务端的签名是否正确，如果不正确则中止。

6）客户签名 T1 并且把签名返回给服务端，服务端广播 T1（如果他们双方有联系的话，每一方都可以广播 T1），币被锁定。

7）客户创建一个新交易 T3，与 T1 相联系，类似于退款交易，有 K1 和 K2 两个输出，把所有币分配给第一个输出 K1，它做了与退款交易相同的事情，但是没有锁定时间，客户

签名 T3，发送给服务端。

8）服务端验证输出到它的地址的币数量是正确的，验证客户提供的签名是正确的。

9）当客户想付钱给服务端时就调整 T3，向服务端的输出增加足够的币，同时减少他自己的币数量，然后重新签名 T3，发送给服务端。客户无须发送整个交易，只需要发送签名和增加的币数量即可。服务端调整 T3 内容与新的币数量相吻合，验证客户的签名并且继续。

整个过程会持续到会话结束，或者到 1 天的时间快到时，此时，AP 会签名和广播最终版本的交易，向它自己分配最终的币数量。退款交易需要处理服务端消息中断和未分配币的情况，如果发生了这些事件，一旦锁定时间过期，客服就可以广播退款交易，拿回所有的钱。

这个协议已经用 bitcoinj 实现了。

当 nLockTime 的交易能够进入内存池、交易替换重新启用时，本协议必须被修改。在这种情况下，无需退款交易。T3 有一个序列号，每次都比前一次大 1，T3 的锁定时间与之前的时间相一致。每次的支付都使得序列号增加 1，以确保会优先发生最后版本。如果没有正常关闭通道协议，则意味着向服务端支付币不会成功，直到锁定时间过期，客户拿回所有币为止。为了避免这种情况，双方共同签名 T3 交易，序列号为 0xFFFFFFFF，导致立即确认，而不用考虑 nLockTime 的值。

锁定时间和序列号可以避免一种攻击，在这种攻击中：AP 提供连通性，客户使用 TX2 的第一版本双花，使币回到他自己的地址中，阻止咖啡屋申报账单。如果客户尝试这么做，该交易不会马上被包括进去，AP 在一段时间内可以观察到该交易被广播，然后广播它看到的最后一个版本，就能推翻客户的双花企图。

后一种协议依赖交易替换，具有更大的灵活性，因为在通道的生命周期中，只要客户能收到服务端的签名，协议就能使客户为自己分配的币的数量越来越少。但是在许多用例中，这个功能是不必要的。交易替换同样允许配置更复杂的超过两方的通道，如何在这种使用场景下详尽地描述协议，就留给读者作为练习吧。

1.5.9　示例 8：多方去中心化彩票

使用示例 6 的一些技术和一些高级的脚本，使得构建无人值守的多方彩票系统成为可能。准确协议在“Secure multiparty computations on Bitcoin”（见参考资料 [12]）中已经有详细描述。

参考资料

[1] http://www.8btc.com/bitcoin_block_chain

[2] http://www.360doc.com/content/14/1228/10/18005502_436322050.shtml

[3] http://www.8btc.com/bitcoin-change-addresses-explanation
[4] http://8btc.com/article-1786-1.html
[5] http://www.8btc.com/bitcoin-transactions
[6] http://8btc.com/article-1775-1.html
[7] http://8btc.com/article-1767-1.html
[8] http://www.8btc.com/understand-bitcoin-script
[9] https://en.bitcoin.it/wiki/Atomic_cross-chain_trading
[10] https://en.bitcoin.it/wiki/Ripple_currency_exchange
[11] https://en.bitcoin.it/wiki/Zero_Knowledge_Contingent_Payment
[12] http://eprint.iacr.org/2013/784

第 2 章 Chapter 2

区块链进阶

2.1 外带数据

区块链的外带数据是指那些保存在区块链上但不进行货币交易的信息，比如需要永久保存的信息，前面提到过存在于区块链上的每一笔交易都有一个输入和输出，区块链的外带数据也是采用类似的方式来存储的。

如果有在比特币上永久存储数据的需求，那么你目前有两种选择，即：“OP_RETURN”和“Multi-Signatures”。OP_RETURN 是指在每个交易的公钥脚本中嵌入 OP_RETURN 操作码，之后放置外带数据；Multi-Signatures 则是指在建立多签地址时，使用空白签名区域来放置数据。此类外带数据的方法旨在让编程者更简单地将数据编码到交易中，同时又不会影响到比特币区块链的功能，目前非常流行，在 Omni、Open Assets、Blockstack 和 Factom 等的区块链上都有应用。

2.1.1 OP_RETURN 外带数据

2013 年，比特币协议中引进了一项新功能，即：创建一种名为 OP_RETURN 的交易，可以嵌入 40 字节小段数据（目前已经是 80 字节长）。最初，这个功能旨在把情境信息加入比特币交易里，比如配送信息等。后来，发展出了更具创造性的用法，即创造最小量的交易（0.00000001 BTC 加上交易费），并且可嵌入任何你想放进去的信息。

比如交易号：

```
495926f46e3aae80088919f363b3b6ff52116e28637b63eb2a681b1fb990d2e4
```

所对应的交易输出为 OP_RETURN 类型，那么它的输出脚本则为：

```
OP_RETURN 4343020549a5710049a57190
```

利用这个功能的第一个有趣的应用就是存在性证明（Proof Of Existence），它可以为任何文件创建一个 Hash，并且可以放入区块链中，这点不同于其他所有文件的身份认证 ID。之后，通过比较区块链里的 Hash 和你手头文件的 Hash，就可以用那个交易的时间戳和存储在其中的 Hash，来证明那个时点上某个文件是确实存在的。只要二者匹配，就有了证明。

值得注意的是，只能在该交易中输出很小金额的比特币，如 0.00005，因为输出到 OP_RETURN 中的比特币是无法被再次使用的。

2.1.2 Multi-Signatures 外带数据

多签名地址是另外一种外带数据的方法。例如，对于 1-of-2 型的多签名地址，我们在建立该地址时，提供的第一个公钥是发送者的，因而输出金额可以赎回；而第二个公钥是空白的，因而该空间可以用来存储外带数据。对数据的要求则是：前缀是数据的长度，后面是用 0 来充值的无数据区域。

该交易的输出脚本为：

```
1 <K1> <0> 2 OP_CHECKMULTISIGN
```

这里的 <0> 区域可用于存储外带数据，该输出金额一样可以被花费。如果要存储更多数据，则可采用 1-of-3 或 1-of-5 型的多签地址。

2.2 Counterparty

合约币 Counterparty 是以合约币协议运行的全套金融工具，合约币协议建立在比特币块链的基础之上，把比特币块链当成可信的时间戳服务和可信的信息发布证明。

目前合约币已经实现了众多的技术创新点，比如燃烧证明、合约、去中心化 XCP 与 BTC 交易所、赌约或期货、资产或股份发行、分红等。Counterparty 是建立在比特币协议上的传输层，用于建立和使用去中心化的财务工具协议。简单来说，可以将 XCP 理解为很多“小的 BTC”即 XCP=“小的 BTC”，但是这些“小的 BTC”（XCP）不仅仅具有货币的交易功能，还具有资产发行（例如发行股票）、股息分配及下注的功能。

每个合约币信息都包括以下特性。

- 有一个源地址。
- 有一个目的地址。
- 有一定数量的比特币，从源代码发送到目的地址（如果存在目的地址的话）。
- 以比特币计的费用，支付给挖到这个交易的矿工。
- 最多 40 字节的外带数据，这些数据通过上述的两个外带数据方法嵌入到比特币交易中。

为了方便区分，每个合约币交易的数据区域都以 UTF-8 格式的 CNTRPRTY 打头，这个

字符串已经足够长，因而很难将合约币的交易与在 OP_RETURN 区域带有伪随机数的比特币交易搞混。在测试情况下（例如：在任何块链中使用 TESTCOIN 合约币网络），这个字符串是“XX”。

识别字节串“CNTRPRTY”之后的四个字节说明了目的地址的存在性、比特币交易费用的多少，以及交易的比特币数量（根据合约消息类型的不同而不同）。其他的数据则根据消息类型具有不同的格式，具体请参考源代码。

另外，每个合约币交易都必须有一个明确且唯一的源地址，在含有合约币交易的比特币交易中，所有输入都必须一致。在比特币交易中资金唯一的源地址，就是合约币交易的源地址。

合约币交易的源地址和目的地址，就是比特币地址，任何比特币地址都可以收到任何合约币的资产（如果该地址有资产的话，也可以向外发送资产）。

此外，需要说明的是，所有信息都会按照顺序进行解析，一次一个，并且会忽略区块边界。

2.2.1　Counterparty 附生链的实现机制详解

附生链支持构建两种类型的交易：BTC 发送和以 BTC 发送合约币的资产分红，这两种交易不包含数据区，对于后者，能使用多个“目的”输出。除 BTC 和 XCP 以外的所有资产均具有如下属性：Asset name、Asset ID、Description、Divisiblity、Callability、Call date（if callable）即赎回日期（如果是可赎回的）、Call price（if callable）即赎回价格（如果是可赎回的）等。

资产名称是大写的 ASCII 字符串，当编码成十进制整数时，其值大于 26^3，小于或等于 256^8。所有资产名称，除了“BTC”和“XCP”以外，必须至少有 4 字节长，而且不能以字符“A”开头。这样处理后，某些 13 字节的资产名称是有效的，但是 14 字节的就不行。

资产可以是可分割的或不可分割的，可分割资产可以分成 8 个十进制的位置。资产可以有描述，也可以随时改变。

资产可以是“可赎回的”，可赎回资产在赎回期之后，可以被现在的发行者，以赎回价格（以 XCP 为单位）强制“赎回”，赎回价格可以设置成该资产首次发行的价格。

可赎回资产可以在赎回日期之后赎回，该赎回日期是在块链中的一个区块中第一次定义的时间。

赎回价格指定为 6 个十制数的精度，是 XCP 与该资产的最小单位的比率。

附生链的交易类型包括：发送（Send）、订单（Order）、BTC 支付（BTCPay）、发行（Issue）、广播（Broadcast）、赌约（Bet）、分红（Dividend）、燃烧（Burn）、取消（Cancel）、回调（Callback）等。下面就来一一介绍下。

2.2.2　发送

发送（Send）是指从源地址发送任何合约币资产到目的地址，如果在解析（以交易顺序）该消息时，发送者还没有足够的资产数量，则该发送消息只能部分满足。Counterparty 支持发送比特币，这里不使用任何数据输出。

2.2.3 订单

订单（Order）是指给定某种资产的特定数量，要得到另外资产的特定数量。“买单”和“卖单”之间没有本质差别。在订单解析时，被给的资产通常会马上被签订合约。也就是说，如果有人想用 1 个 XCP 换 2 个 BTC，一旦他发布了这个订单，那么他的 XCP 账号马上会减去 1 个 XCP。

当订单在块链中可见时，协议会将它与另一个以前见过的开放订单撮合在一起。两个被撮合成功的订单称为“订单对”，如果订单对中的任何一个订单包含比特币，那么这个订单对会被指定为“待处理”状态，直到必要的 BTCPay 交易发布；如果订单对中的订单没有包含比特币，那么这个贸易将立即完成，并且以协议自身指定的地址形成新的收支平衡。

所有订单都是定价单，询价指定了一个人想要付出和得到的比率，订单会匹配定价以下的最高价格，订单对就是按照这个价格撮合的。也就是说，如果有个开放订单以 0.11XCP/ 每份资产卖出，第二个卖单以 0.12XCP/ 每份资产卖出，第三个卖单以 0.145XCP/ 每份资产卖出，然后有一个新订单要以 0.14XCP/ 每份资产的价格买入，那么它将会先匹配第二个买单，XCP 和 BTC 将会以 0.12XCP/ASST 的价格成交。

所有订单允许部分执行，也就是说，订单并非要么完全成交，要么不成交。在前一个例子中，如果购买比特币的一方想买的数量多于第一个卖单的数量，那么买单剩余未成交的数量会由后面的现存卖单来满足。在所有可能的订单对撮合完之后，当前的买单被列为开放订单（如果还有数量未被满足的话），如果存在多个价格相同的开放买单，则订单将按照时间顺序撮合。

用户发布开放订单之后，开放订单会在一定数量的区块后过期，当订单过期时，所有担保的资金都会返回到订单发布的那一方。

等待比特币支付的订单对将在 10 个区块后过期，其中的订单将会重新发布。一般情况下，不会存在虚假交易，因为每一方提供的资产都存储在担保处。然而，担保比特币是不可能的，因而那些想购买比特币的人会要求只匹配有向比特币矿工支付交易费用的交易。另一方面，当创建订单售出比特币时，用户可以支付他愿意支付的任意费用；如果是部分订单，则仅支付部分费用。

此外，可用支付比特币的方式来关闭那些等待 BTCPay 消息的订单对。在 BTCPay 消息的数据区块中，存储着两个 Hash 串连接而成的字节串，这两个 Hash 串是由订单对中的两个订单 Hash 生成的。

2.2.4 发行

资产可以以发行消息类型的方式进行发行（Issue）：用户指定名称和数量，协议计入相应的地址。资产名称必须是唯一的，或者以前被相同地址发行过的。如果要重新发行一个资产，也就是说，对已经发行的资产进行增发，那么发行的资产名称、可分割性和发布地址必须匹配。对某个已经存在的资产进行增发，这个权利可以转移给别的地址。资产可以被不可

逆地锁定，防止进行增发，以保证资产拥有者免受通胀风险。

2.2.5　广播

广播（Broadcast）用于发布文本或数字信息，并且附带一个时间戳作为系列广播的一部分，这系列广播被称为“反馈”。一个反馈和一个地址相关联：从给定地址过来的任何广播，都是该地址反馈的一部分。一个反馈的时间戳必须是单向增加的。赌注以数字形式在反馈中下注，这个数值可以是货币的价格，或者可以是对未来事件可能的离散输出的一部分描述。例如，有人可能以文本形式这么描述：“US QE on 2014-01-01: dec = 1, const = 2, inc = 3”，并且宣布结果是“US QE on 2014-01-01: decrease!”和数值 1，更为复杂的赌约可以以非块链的方式进行发布。

发布内容为文本字符串“LOCK”（大小写不敏感）的单个广播可以锁定反馈，阻止它成为以后的广播源地址，同时也阻止它成为任意新赌约的主题（如果反馈被锁定，但还有开放的或未解决的赌约与之相关，那么，那些赌约或赌约对会无损害地过期）。广播 -1 的数值会被赌约结算所忽略。反馈以它的发布地址来识别。

2.2.6　赌约

赌约（Bets）是指打赌一个特定反馈会等于（或者不等于）某个特定时间的一个目标值，例如，打赌 2020 年 1 月 1 日 12 点整比特币的价格是否为 $10 000 美元。参与赌博的人用他们的赌金进行担保，当到达指定时间，该赌约就会按照反馈结果进行结算，赢者获得赌金。

等于 / 不等于赌约不能用杠杆。然而，为了让两个赌注能被撮合，它们的杠杆水平、时间期限和目标值都必须相同，否则，它们会以订单的形式撮合，除非赌约的赔率与订单价格是反相关的（赔率 = 保证金 / 庄家保证金），如果有可能，每个赌约都会以尽可能高的赔率撮合到开放赌约。

目标值必须非负，赌约对（合约）不受 -1 的广播值所影响。赌约的最后期限不能晚于它们指定反馈的最后一个广播的时间戳。赌约的过期与订单相同，例如：过了特定的区块数量后就会过期。如果赌约对在 2016 个区块后，发现一个区块的时间戳在赌约的最后期限之后过期，那么，赌约费用将会是初始保证费用的一定比例，这部分并非赌约收入。

因为区块时间的存在，以及交易在块链中被订单化这种非确定性方式的存在，所有合约必须非增量解决，但是涉及的资金必须马上放入担保契约，而且必须有结算日期。否则，有人看到价格下跌了，就会把要被扣除的资金隐藏起来。

2.3　挖矿算法解析

2.3.1　PoW 挖矿算法及分析

PoW（Proof of Work），即工作证明。也就是说，你获得多少货币，取决于你挖矿贡献的

有效工作，比如，你的电脑性能越好，分给你的矿就会越多，即根据你的工作证明来执行货币的分配。大部分的虚拟货币，比如比特币、莱特币等，都是基于 PoW 模式的虚拟货币（算力越高、挖矿时间越长，你获得的货币就会越多）。

第 1 章已经说过挖矿算法其实就是通过一个 Hash 函数找到一个满足当前难度的 Nonce（包含在区块头里面）的值，Hash 函数输入数据的长度是任意的，将产生一个固定且绝不雷同的值，可将其视为输出的数字指纹。对于特定输入，Hash 的结果每次都不一样，任何实现相同 Hash 函数的人都可以计算和验证。加密 Hash 函数的主要特征就是不同的输入几乎不可能出现相同的数字指纹。因此，相对于随机的选择输入，有意地选择输入去找一个特定的 Hash 值，这几乎是不可能的，否则就破解了所使用的 Hash 算法，如 SHA-256 等。

矿工用一些交易来构建候选区块，他会计算这个区块头 Hash 的值，看其是否小于当前目标值，如果这个值小于目标值，矿工就会修改这个 Nonce 的值，然后再试一次。通常来说一个矿工会做成千上万次 Hash 运算，从而得到一个合适的 Nonce 的值，使得区块头 Hash 满足当前难度。这也是 PoW（工作量证明）算法的由来。

PoW 要求出示一定的证明表明工作量，证明可以是直接记录也可以通过概率表示，其中对于由小概率事件累计而成的工作，出示结果等同于证明了工作量（因为不太可能直接得到小概率结果）。在比特币和其他类比特币的系统中，PoW 系统是以合乎要求的 Hash 作为工作结果的。由于矿工需要一定量的计算才能取得合法的计算结果，因此得到合法的计算结果就可以证明完成了一定量的计算。

2.3.2 PoS 股权证明算法及分析

PoS（Proof of Stake），即股权证明。它又是什么意思呢？简单来说，它是根据你持有货币的量和时间，给你发利息的一个制度。在股权证明模式下，有一个名词叫币龄，每个币每天产生 1 币龄，例如你持有 100 个币，总共持有了 30 天，那么，此时你的币龄就为 3000，这个时候，如果你发现了一个 PoS 区块，你的币龄就会被清空为 0。你每被清空 365 币龄，你将会从区块中获得 0.05 个币的利息（可以理解为年利率 5%），那么在这个案例中，利息 = 3000 × 5%/365 = 0.41 个币。

以现有的比特币运行发展情况来看，比特币每年的挖矿产量都在不断减半，我们可以预计，随着比特币产量的不断降低，矿工人数也会越来越少，这样就会导致整个比特币网络的稳定性出现问题。PoS 的解决方案是鼓励大家都去打开钱包客户端程序，因为只有这样才可以发现 PoS 区块，才会获得利息，这也增加了网络的健壮性。

其次还有一个担忧，就是当矿工人数降低时，比特币很可能会被一些高算力的人或团队进行 51% 攻击，如果采用 PoS 体系，你即便拥有了全网 51% 的算力，也未必能够进行 51% 攻击，因为这还要求攻击者持有全球 51% 的货币量，而这是很难达到的。

现在，我们知道比特币是一个永远不会膨胀的体系了，因为它的货币总量看起来貌似是固定的，但实际上比特币是一个货币紧缩的体系，因为总存在钱包永久丢失的可能，PoS 采

用类似年利率的方式在一定程度上是可以缓解这个问题的。

不过，到目前为止，PoS 算法还没有被比特币采用的迹象。

2.3.3　DPoS 股份授权证明算法及分析

股份授权证明机制（DPoS）是一种新的保障加密货币网络安全的算法，由比特股 bitshares 提出。它在尝试解决比特币采用的传统工作量证明机制（PoW）及点点币和 NXT 所采用的股份证明机制（PoS）的问题的同时，还能通过实施科技式的民主，来抵消中心化所带来的负面影响。

以比特股为例来说，DPoS 机制是让每一个持有 BTS（比特股发行的一种加密货币）的人对为整个系统资源当代表的人进行投票，而获得票数最多的 101 个代表将进行交易打包计算。对此，可以理解为有 101 个矿池，这 101 个矿池彼此的权利是完全对等的。那些握着 BTS 选票的人可以随时通过投票更换这些代表（矿池），但如果他们提供的算力不稳定，计算机宕机或试图利用手中的权力作恶，那将会立刻被愤怒的选民们踢出整个系统，而后备代表可以随时顶上去。从某种角度来看，DPoS 有点像美国的议会制度，只不过不是四年一次选举，而是时刻都在选举中。

2.4　Sidechains

Sidechains（侧链）实质上不是指特定的某个区块链，而是指那些遵守侧链协议的所有区块链，这个词是针对比特币主链来说的。侧链协议是指可以让比特币安全地从比特币主链转移到其他区块链，同时又可以让其他区块链上的货币安全返回到比特币主链的一种协议。

所以从某种程度上来说，现在市面上的所有区块链，例如以太坊、莱特币、狗狗币等区块链都可以成为侧链应用。侧链的实现具有重大意义，它意味着比特币可以在不同的区块链上流通，其应用范围和应用前景会更加广泛。一旦侧链的应用流行起来，有创意有想法的人就会研发出各种不同的侧链协议来和比特币进行对接，很显然这种方式会进一步巩固比特币在区块链中的地位。

2.4.1　侧链背景

侧链的提出主要是基于以下几个原因。

（1）应对其他区块链的应用威胁

现在市面上出现了几种非常流行的区块链，例如以太坊区块链、比特股区块链等，这些不断产生的新的区块链势必会对比特币区块链产生很大的威胁。以太坊区块链更是提出了智能合约这个有望颠覆整个区块链的应用，而到目前为止，基于比特币的应用开发项目还不多，已有的项目难度也很大。

（2）比特币核心开发组不欢迎附生链

比特币现有的应用中已经存在合约币和彩色币等附生链应用，但是基于一些原因比特币核心开发组并不是很欢迎这些应用。他们的考虑是这些应用会在一定程度上降低比特币区块链的安全性。

（3）Blockstream 商业化的考虑

有了以太坊众筹的前车之鉴，比特币核心开发组也希望能以某种商业化的方式来实现更高的回报，所以基于比特币的侧链应用也激发了一群人开始尝试商业化这些应用。

基于以上三个原因，提出侧链协议，把比特币从主链上安全地转移到了其他货币区块链，这样既增加了区块链的多样性又可以应对二代币的竞争，同时这些应用本身确实具有比较大的商业前景，一旦实现商业化，这一块也将成为一块很大的蛋糕。

2.4.2 技术原理

楔入式侧链技术（pegged sidechain），它将实现比特币和其他数字资产在多个区块链间的转移，这就意味着用户们在使用他们已有资产的情况下，就可以访问新的加密货币系统。目前，侧链技术主要由 Blockstream 公司负责开发。

这里先列出“侧链”所需具备的属性，具体如下。

- 在侧链间移动的资产应当能够被当前持有者移回，但除此之外的任何人都不行（包括前持有者）。
- 资产的移动应当是无交易对手风险的；也就是说，不诚实的一方无法阻止转移的发生。
- 资产转移应当是元操作（原子操作）的，即要么完全完成，要么根本不发生。不存在会导致资产损失或允许欺诈产生的失败模式。
- 侧链应当设有防火墙：一个会使某条链发生资产铸造（或偷盗）的缺陷（Bug），不应导致其他任何链出现资产的铸造或偷盗。
- 区块链重组时应当处理干净，即使是在资产转移的期间也要如此；任何破坏应当只发生在它所处的侧链上。总的来说，理想情况下，侧链应当完全独立，其他链上所需的全部数据应由用户提供。侧链的验证者应当只有在侧链本身的显式共识规则有要求时，才去跟踪其他链。
- 不应要求用户去跟踪他们未主动使用的侧链。

早期“转移”钱币的解决方案是用一个可公开识别的方式来销毁比特币的，新的区块链能够检测到，以允许铸造新币［ Bac13b ］。这解决了上面提到的部分问题，但由于这种方法只允许单向转移，因此还不足以满足我们的目的。我们提出的方案是由资产转移的交易本身提供所有者证明，从而实现资产转移，以避免让节点有跟踪发送方链的需求。从上层实现的角度来说，当资产从一个区块链向另一个链移动时，我们在第一个区块链上创建交易锁定资产，然后在第二个区块链上创建一笔交易，该交易的输入中包含一个锁定已正确完成的密码学证明。这些输入可用某种资产类型来标记，比如创生出资产的区块链的创世哈希（genesis hash）。

第一个区块链称之为父链，第二个则简称为侧链。在某些模型中，两条链可对称地来处理，因此这一术语是相对而言的。如果打算将资产从（初始）父链转移到一条侧链，期间可能会再转移到别的侧链，最终还能转回至父链，并保全初始资产。一般情况下，我们把父链看成是比特币系统，侧链则是其他区块链中的某一个。当然，侧链的币（coin）也可以在侧链间传递，并非只能与比特币系统进行往来；不过，由于任何一个最初从比特币系统移动的币都可以移回去，所以不管变成什么样，它仍是个比特币。

此外，由于侧链是从父链中转移现有资产的而不是另铸新资产，因此，侧链不会引起未经授权的铸币，维护资产的安全和稀缺性是依靠父链来实现的。

更进一步地说，参与者不必再担心他们的持有物会被一个实验性竞争链锁住，因为侧链币能够用等额的父链币来赎回。这就提供了一个退出机制，减少因无人维护软件而造成的损失。

前面已经提到楔入式侧链技术，它的双向挂钩（2WP）允许将比特币从比特币区块链转移到辅助区块链，反之亦然。“转移”实际上是一种错觉：比特币其实并没有转移，只是在比特币区块链上被暂时锁定了，而且同时在辅助区块链上有相同数量的等价令牌被解锁。当等量的令牌在辅助区块链上被再次锁定时，原先的比特币就会被解锁。这实质上就是双向挂钩所要实现的功能。但这一功能也存在一个问题，即理论上只有当辅助区块链最终结算时才能实现这一功能。因此，任何双向挂钩系统都必须做出妥协，并且依靠于双向挂钩的相关参与者都是诚实的这一假设。最重要的假设是，主要的区块链是无须审查的，而且大多数比特币矿工都是诚实的。另外可能需要的一个假设是，大多数监管锁定比特币的第三方也是诚实的。如果这些假设不成立，则比特币及等效辅助区块链的令牌就有可能被同时解锁，那么恶意的双花就变得可行了。任何双向挂钩系统都必须选择一种措施，使得被假设要诚实的各方都能在经济和法律方面被鼓励依章办事。这包括分析这些关键方对区块链网络进行攻击的成本及后果。双向挂钩实施的安全性取决于激励机制，以便参与双向挂钩系统的关键方能够真正执行双向挂钩所应实现的功能，如图 2-1 所示。

双向锚定分为四个阶段，具体如下。

1）发送锁定交易，把比特币锁定在比特币主链上。

2）等待确认期。确认期的作用是等待更多区块确认锁定交易，可防止假冒锁定和拒绝服务攻击，等待时间是 1 ~ 2 天。

3）在侧链上赎回比特币。上述确认期结束后，用户在侧链上创建一个交易花掉锁定交易的输出，并且提供一个 SPV 工作量证明，输出到自己侧链上的地址。这个交易也称为赎回交易，SPV 工作量证明是指赎回交易所在区块的工作量证明。

4）等待一个竞争期。竞争期的作用是防止双重支付，在此期间，新转移过来的币不能在侧链上花费。竞争期的目的是防止重组时出现双花，在重组期间会转走先前锁定的币。在这个延迟期内的任何时刻，如果有一个新的工作证明发布出来，且对应的有着更多累计工作量的链中没有包含那个生成锁定输出的区块，那么该转换将被追溯为失效。我们称此为重组证明。

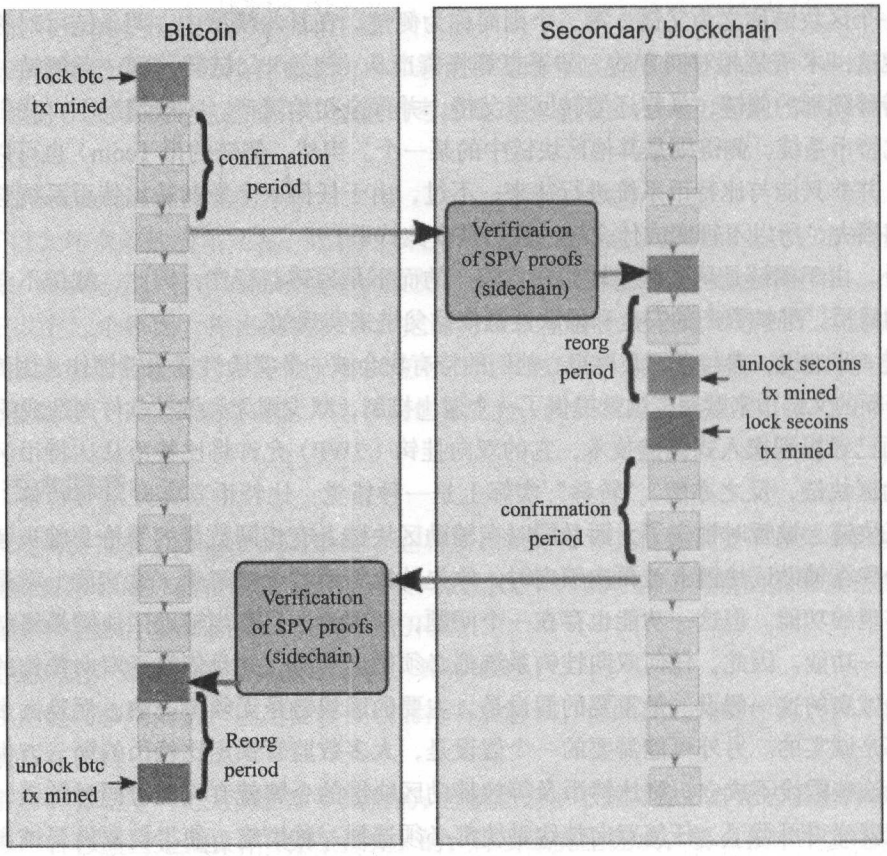


图 2-1 双向锚定示意图

竞争期结束后，这个赎回交易将被打包到区块中，用户就可以使用自己的比特币了，从侧链转移比特币的过程也是如此，当用户想把币从侧链上转回父链时，与原先转移所用的方法相同：在侧链上将币发送至一个 SPV 锁定的输出，并产生一个充分的 SPV 证明来表明该输出已经完成，然后使用这个证明来解锁父链上先前被锁定的那个等面值的输出。

由于楔入式侧链可能会从很多链中搬运资产，且无法对这些链的安全性做出假定，因此，不同资产不可相互兑换是非常重要的（除非是一个显式声明的交易）。否则，恶意用户可以通过创建一条资产毫无价值的无价值链进行偷盗，即将这样一种资产移到一个侧链，再用它去兑换别的东西。为了应对这种情况，侧链必须有效地将不同父链中的资产处置为不同的资产类型。

总之，我们提议让父链和侧链相互做数据的 SPV 验证。由于不能指望父链客户端能看到每条侧链，因此为了证明所有权，用户必须从侧链导入工作量的证明到父链。在对称式双向楔入中，反向的操作也是如此。

为了让比特币系统成为父链，需要有一个能识别和验证 SPV 证明的脚本扩展。最起码

的要求是，这种证明需要做得足够小，以便能放进比特币系统一个交易之中。不过，这只是一个软分叉，对于不使用新功能的交易不会产生影响。

2.5　最新比特币技术

随着区块链的不断发展，区块链技术也在飞速发展，不仅已经有了相对成熟的应用，还有一些原型应用也被提出，下面将介绍几个现在比较流行的在区块链中的应用。

2.5.1　IBLT

比特币系统（Bitcoin）需要矿工（或者说矿池）及全节点的分散化，以实现某些人认为的比特币核心属性：抗审查性（censorship resistance）。因此，区块大小的争议也意味着是一种权衡。更大的区块，允许比特币网络可以承载更多的交易，但也会带来更多的问题，它需要更多的时间来传播交易，虽然这有利于大矿工和矿池，但同时，增加的数据传递对于用户运行全节点而言，也是一种打击。

幸运的是，当前也存在着一些提议，它们可以提高比特币网络的效率，并降低更大区块将会产生的风险。而其中最有前途的创新，就是可逆式布鲁姆查找表（Invertible Bloom Lookup Table），或者简称为 IBLT，如图 2-2 所示。

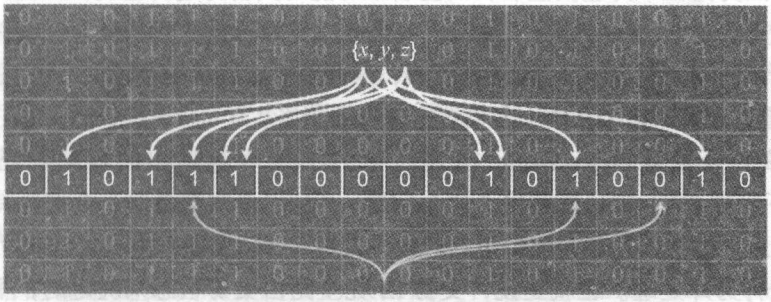


图 2-2　IBLT 原理示意图

这一概念首先是由 Bitcoin Core 和 Bitcoin XT 开发者加文·安德烈森（Gavin Andresen）提出的，那么这个可逆式布鲁姆查找表解决了什么问题呢？

通常情况下，所有的比特币交易，都是通过节点到节点来完成对等式网络交易的，然后通过个人节点的内存池（mempool，记录未确认的交易）进行存储。当一位矿工发现一个区块时，它包括（部分）区块中的交易，随后便会在同一对等网络中广播该区块。当然，这也意味着，区块中的所有交易，都有效地在网络上发送了两次：一次作为一笔交易，另一次则作为区块的一部分。可见，在区块中已经有了冗余。不过，大多数节点已经知道该区块包含的一些内容了。因为它们已经看到过了，如果我们能够优化它，那么就可以加快区块的广播

速度。这也降低了中心化的压力，因为矿工们可以更快地将他们的区块弄出来。

其基本原理如下：首先，在一个区块中包含的所有交易，都会写入一个表（table）中，每一笔交易均会始于表上每一个不同的点。然而，存在的交易数远多于表的空间（room），所以它会导致令人绝望的重叠结果。这使得 IBLT 显得非常密集，但对于那些无法访问任何交易数据的人而言，IBLT 是无法读取的，也是无法破译的。而那些拥有交易数据的人，则可以通过使用类似的逻辑，将他自己的交易填充到一个 IBLT 中，然后比较 IBLT 上的重叠交易数据。如果两个 IBLT 最终看起来完全一样，这就意味着所有的交易都是完全匹配的。即使这两个 IBLT 最终看起来并不是完全相同，但只要交易集是非常相似的，这就可能仍然是有用的。在这种情况下，这两个 IBLT 可以进行比较，采用这种方式，所有相同的交易就可以抵消掉一方。而 IBLT 中“剩余的”交易，往往可以用于重构丢失的交易。

2.5.2 隔离见证

每一个比特币交易，都可以分为两部分。第一部分是转账记录，第二部分是用来证明这个交易合法性（主要是签名）的。第一部分可称为“交易状态”，第二部分就是所谓的“见证”（witness）。如果你只关心每个账户的余额，那么转账记录就已经足够。只有部分人（主要是矿工）才有必要取得交易见证。

中本聪设计比特币时，并没有把两部分资料分开处理，因此导致交易 ID 的计算混合了交易和见证。因为见证本身包括签名，而签名不可能对其自身进行签名，因此见证可以由任何人在未得到交易双方同意的情况下进行改变，造成所谓的交易可塑性（malleability）。在交易发出后、确认前，交易 ID 可以被任意更改，因此基于未确认交易的交易是绝对不安全的。在 2014 年就曾有人利用这个漏洞大规模攻击比特币网络。

比特币核心开发员 Pieter Wuille 在 2015 年 12 月于香港提出的隔离见证（Segregated Witness，以下简称 SW）软分叉解决了这个问题。SW 用户在交易时，会把比特币传送到有别于传统的地址。当要使用这些比特币的时候，其签名（即见证）并不会记录为交易 ID 的一部分，而是进行另外处理。也就是说，交易 ID 完全是由交易状态来决定的，不会受见证部分的影响。这种做法有如下几个重要的结果。

- 可以用软分叉增加最大区块容量。因为旧节点根本看不到这些被隔离的见证，即使真实的区块已经超过了 1MB，它们仍会以为没有超过限制而会接受区块。在整场有关区块容量的辩论中，最大的难点就是硬分叉。SW 可以提供约 2MB 的有效区块空间而没有任何硬分叉风险。
- 从此以后，只有发出交易的人才可以改变交易 ID，没有任何第三方可以做到。如果是多重签名交易，那么只有得到多名签名人的同意才能改变交易 ID。这可以保证一连串的未确认交易的有效性，是双向支付通道或闪电网络所必须具备的功能。有了双向支付通道或闪电网络，二人或多人之间就可以从实际上进行无限次交易，而无须把大量零碎交易放在区块链上，从而大大降低区块空间压力。

- 轻量钱包可以变得更轻量，因为它们无须再接收见证数据。
- 可以大幅改善签名结构。在区块链上，曾经有一个超过 5000 个输入的交易，因为签署设计缺憾，需要半分钟才能完成检查。SW 软分叉解决了这个问题。

2.5.3　闪电网络

比特币从诞生之日起就一直存在着若干个技术问题，比如交易处理能力，目前全网每秒只有 7 笔交易；其次是区块产生时间，现在大致是每 10 分钟出一个块；再其次是交易确认问题，一般建议在等待 6 个区块后才视为交易被确认，大额交易则建议等待更长时间；最后是容量，目前已经生成了 40 多万个区块，约 60GB 的数据量，且眼见的未来中只见增加不见减少。这些无疑会是区块链技术在实际应用中的一大障碍。

在闪电网络出现之前，虽然比特币社区也试图通过区块扩容、隔离见证等技术在一定程度上提高交易处理的能力，但这些方式并不能导致交易处理能力出现数量级的改善。至于前面提及的其他技术难题，也不是短时间内就能攻克的，比如，现存的 PoW 机制是万万动不得的，需要等待多个区块的确认也是不能触碰的底线，更麻烦的是，交易处理能力和区块链数据容量似乎是一对无可调和的矛盾。

因此，闪电网络提供了一个可扩展的微支付通道网络。交易双方若在区块链上预先设有支付通道，就可以多次、高频、双向地通过轧差方式实现瞬间确认的微支付。双方若无直接的点对点支付通道，只要网络中存在一条连通双方的、由多个支付通道构成的支付路径，则闪电网络也可以利用这条支付路径实现资金在双方之间的可靠转移。

闪电网络并没有试图解决单次支付的银货对付问题，其假设是单次支付的金额足够小，即使一方违约另一方的损失也非常小，风险可以承受。因此使用时必须注意“微支付”这个前提。多少资金算“微”，显然应该根据业务而定。

2.5.4　RSMC

闪电网络的基础是交易双方之间的双向微支付通道，RSMC（Recoverable Sequence Maturity Contract）定义了该双向微支付通道的最基本工作方式。

微支付通道中沉淀了一部分资金，也记录有双方对资金的分配方案。通道刚设立时，初始值可能是 {Alice: 0.4, Bob: 0.6}，这意味着打入通道的资金共有 1.0 BTC，其中 Alice 拥有 0.4 BTC，Bob 拥有 0.6 BTC。通道的设立会记录在比特币区块链上。

假设稍后 Bob 决定向 Alice 支付 0.1 BTC。双方在链下对最新余额分配方案 {Alice: 0.5, Bob: 0.5} 进行签字认可，并签字同意作废前一版本的余额分配方案 {Alice: 0.4, Bob: 0.6}，Alice 实际上就获得了 0.5 BTC 的控制权。如图 2-3 所示。

如果 Alice 暂时不需要将通道中现在属于她的 0.5 BTC 用作支付，那么她可以无须及时更新区块链上记录的通道余额分配方案，因为很可能一分钟后 Alice 又需要反过来向 Bob 支付 0.1 BTC，此时他们仍然只需要在链下对新的余额分配方案达成一致，并设法作废前一版

本轮余额分配方案就行了。

类型	冻结	Alice	Bob
无条件	1.0	0.4	0.6

类型	冻结	Alice	Bob
无条件	1.0	0.5	0.5

图 2-3 RSMC 账户余额分配示意图

如果 Alice 打算终止通道并动用她的那份资金，那么她可以向区块链出示双方签字的余额分配方案。如果一段时间之内 Bob 不提出异议，那么区块链会终止通道并将资金按协议转入各自预先设立的提现地址。如果 Bob 能在这段时间内提交证据证明 Alice 企图使用的是一个双方已同意作废的余额分配方案，则 Alice 的资金将被罚并给到 Bob。

2.5.5 HTLC

RSMC 只支持最简单的无条件资金支付，HTLC（Hashed Timelock Contract）则进一步实现了有条件的资金支付，通道余额的分配方式也因此变得更为复杂。

通过 HTLC，Alice 和 Bob 可以达成这样一个协议：协议将锁定 Alice 的 0.1 BTC，在时刻 T 到来之前（T 以未来的某个区块链的高度来表述），如果 Bob 能够向 Alice 出示一个适当的 R（称为秘密），使得 R 的 Hash 值等于事先约定的值 H(R)，Bob 就能获得这 0.1 BTC；如果直到时刻 T 过去 Bob 仍然未能提供一个正确的 R，那么这 0.1 BTC 将自动解冻并归还 Alice。

由于到期时间 T、提款条件 H(R)、支付金额和支付方向的不同，同一个通道上可以同时存在多个活动的 HTLC 合约，再加上唯一的通过 RSMC 商定的无条件资金余额，余额分配方式会变得相当复杂。下面以图 2-4 为例来讲解这一大致流程。

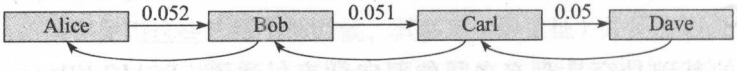


图 2-4 HTLC 交易流程示意图

如图 2-4 所示，Alice 想给 Dave 发送 0.05 BTC，但 Alice 和 Dave 之间并没有微支付通道。Alice 找到了一条经过 Bob、Carol 到达 Dave 的支付路径，该路径由 Alice/Bob、Bob/Carol 和 Carol/Dave 这样三个微支付通道串接而成。

此时，Dave 生成了一个秘密 R 并将 Hash(R) 发送给 Alice，Alice 不需要知道 R。

该交易的流程具体如下。

1）Alice 和 Bob 商定一个 HTLC 合约：只要 Bob 能在“3”天内向 Alice 出示 Hash 正确的 R，Alice 就会支付 Bob 0.052 BTC；如果 Bob 做不到，这笔钱将 3 天后自动退还 Alice。

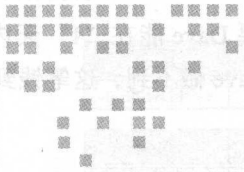
2）同样地，Bob 和 Carol 商定一个 HTLC 合约：只要 Carol 能在“2”天内向 Bob 出示 Hash 正确的 R，Bob 就会支付 Carol 0.051 BTC；如果 Carol 做不到，这笔钱到期后将自动退

还 Bob。

3）最后，Carol 和 Dave 商定一个 HTLC 合约：只要 Dave 能在“1”天内向 Carol 出示 Hash 正确的 R，Carol 就会支付 Dave 0.05 BTC；如果 Dave 做不到，这笔钱到期后将自动退还 Carol。

参考资料

[1] http://www.8btc.com/walletla
[2] http://tech.huanqiu.com/news/2014-12/5282805.html
[3] http://www.8btc.com/counterparty_protocol
[4] http://www.zhihu.com/question/22369364/answer/21169413
[5] http://www.wanbizu.com/baike/201410293172.html
[6] http://www.8btc.com/sidechains-drivechains-and-rsk-2-way-peg-design
[7] http://www.8btc.com/?p=74026
[8] http://toutiao.com/i6237702550993240578/
[9] http://www.8btc.com/ln-rn-corda



Chapter 3 第 3 章

密码学基础

3.1 Hash 函数

Hash 函数是密码学的一个重要分支，它是一种将任意长度的输入变换为固定长度的输出且不可逆的单向密码体制。Hash 函数在数字签名和消息完整性检测等方面有着广泛的应用。

3.1.1 技术原理

Hash 函数又称为哈希函数、散列函数、杂凑函数。它是一种单向密码体制，即一个从明文到密文的不可逆映射，只有加密过程，没有解密过程。

Hash 函数可以将满足要求的任意长度的输入进行转换，从而得到固定长度的输出。这个固定长度的输出称为原消息的散列值（Hash Value）或消息摘要（Message Digest）。Hash 函数的数学表述为：

$$h = H(m)$$

其中，H 是 Hash 函数，没 m 是任意长度明文，h 是固定长度的 Hash 值。

理想的 Hash 函数对于不同的输入可以获得不同的 Hash 值。如果 x, x' 是两个不同的消息，存在 $H(x) = H(x')$，则称 x 和 x' 是 Hash 函数 H 的一个碰撞。

由于 Hash 函数这种单向的特征及长度固定的特征使得它可以生成消息或数据块的消息摘要（也称为散列值、哈希值），因此在数据完整性和数字签名领域有着广泛的应用。

典型的 Hash 函数有两类：消息摘要算法（MD5）和安全散列算法（SHA）。

Hash 函数具有如下特点。

❑ 易压缩：对于任意大小的输入 x，Hash 值 $H(x)$ 的长度很小，在实际应用中，函数 H

产生的 Hash 值其长度是固定的。

- 易计算：对于任意给定的消息，计算其 Hash 值比较容易。
- 单向性：对于给定的 Hash 值 h，要找到 m' 使得 $H(m')=h$ 在计算上是不可行的，即求 Hash 的逆很困难。
- 抗碰撞性：理想的 Hash 函数是无碰撞的，但在实际算法的设计中很难做到这一点。有两种抗碰撞性：一种是弱抗碰撞性，即对于给定的消息 x，要发现另一个消息 y，满足 $H(x)=H(y)$ 在计算上是不可行的；另一种是强抗碰撞性，即对于任意一对不同的消息 (x,y)，使得 $H(x)=H(y)$ 在计算上也是不可行的。
- 高灵敏性：这是从比特位角度出发的，指的是 1 比特位的输入变化会造成 1/2 的比特位发生变化。

3.1.2　SHA-1 算法

1993 年美国国家标准技术研究所 NIST 公布了安全散列算法 SHA-0 标准，1995 年 4 月 17 日公布的修改版本称之为 SHA-1。SHA-1 在设计方面很大程度上是模仿 MD5 进行设计的，但它对任意长度的消息均是生成 160 位的消息摘要（MD5 仅仅生成 128 位的摘要），因此抗穷举搜索能力更强。它有 5 个参与运算的 32 位寄存器字，消息分组和填充方式与 MD5 相同，主循环也同样是四轮，但每轮要进行 20 次操作，包含非线性运算、移位和加法运算等，但非线性函数、加法常数和循环左移操作的设计与 MD5 有一些区别。

SHA-1 的输入是最大长度小于 2^{64} 位的消息，输入消息以 512 位的分组为单位进行处理，输出是 160 位的消息摘要。SHA-1 具有实现速度高、容易实现、应用范围广等优点，其算法描述如下。

1）对输入的消息进行填充：经过填充后，消息的长度模 512 应与 448 同余。填充的方式为第一位是 1，余下各位都为 0。再将消息被填充前的长度以 big-endian 的方式附加在上一步留下的最后 64 位中。该步骤是必须的，即使消息的长度已经是所希望的长度。填充的长度范围是 1 到 512。

2）初始化缓冲区：可以用 160 位来存放 Hash 函数的初始变量、中间摘要及最终摘要，但首先必须进行初始化，对每个 32 位的初始变量赋值，即：

$$H_0 = 0x67452301$$
$$H_1 = 0xefcdab89$$
$$H_2 = 0x98badcfe$$
$$H_3 = 0x10325476$$
$$H_4 = 0xc3d2e1f0$$

3）进入消息处理主循环，处理消息块：一次处理 512 位的消息块，总共进行 4 轮处理，每轮进行 20 次操作，如图 3-1 所示。这 4 轮处理具有类似的结构，但每轮所使用的辅助函数和常数都各不相同。每轮的输入均为当前处理的消息分组和缓冲区的当前值 A、B、C、D、E，

输出仍放在缓冲区以替代旧的 A、B、C、D、E 的值。第四轮的输出再与第一轮的输入 CV_q 相加，以产生 CV_{q+1}，其中加法是缓冲区 5 个字中的每个字与 CV_q 中相应的字模 2^{32} 相加。

4）输出：所有的消息分组都被处理完之后，最后一个分组的输出即为得到的消息摘要值。

SHA-1 的步函数如图 3-2 所示，它是 SHA-1 最为重要的函数，也是 SHA-1 中最关键的部件。

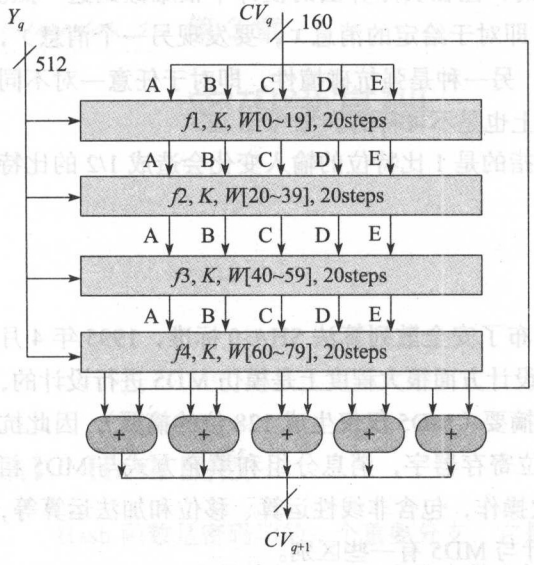


图 3-1 单个 512 位消息块的处理流程

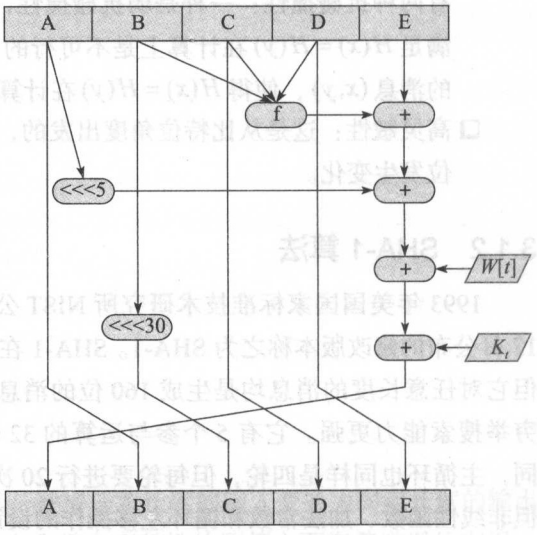


图 3-2 SHA-1 的步函数

SHA-1 每运行一次步函数，A、B、C、D 的值就会依次赋值给 B、C、D、E 这几个寄存器。同时，A、B、C、D、E 的输入值、常数和子消息块在经过步函数运算后就会赋值给 A。

$$A = (ROTL^5(A) + f_t(B,\ C\quad D) + E + W_t + K_t) \bmod 2^{32}$$

$$B = A$$

$$C = ROTL^{30}(B) \bmod 2^{32}$$

$$D = C$$

$$E = D$$

其中，t 是步数，$0 \leqslant t \leqslant 79$，$W_t$ 是由当前 512 位长的分组导出的一个 32 位的字，K_t 是加法常量。

基本逻辑函数 f 的输入是 3 个 32 位的字，输出是一个 32 位的字，其函数表示如下。

当 $t = 0 \sim 19$ 时，

$$f_t(B,C,D) = (B \wedge C) \vee (\overline{B} \wedge D)$$

当 $t = 20 \sim 39$ 时，

$$f_t(B,C,D) = B \oplus C \oplus D$$

当 $t = 40 \sim 59$ 时，

$$f_t(B,C,D)=(B\wedge C)\vee(B\wedge D)\vee(C\wedge D)$$

当 $t=60\sim79$ 时，

$$f_t=(B,C,D)=B\oplus C\oplus D$$

其中 $\wedge$、$\vee$、$-$、$\oplus$ 分别是与、或、非、异或 4 个逻辑运算符。

对于每个输入分组导出的消息分组 W_t，前 16 个消息字 $W_t(0\leqslant t\leqslant 15)$ 即为消息输入分组对应的 16 个 32 位字，其余 $W_t(16\leqslant t\leqslant 79)$ 可按如下公式得到：

$$W_t=ROTL_1(W[t-16]\oplus W[t-14]\oplus W[t-8]\oplus W[t-3])$$

其中，$ROTL_s$ 表示左循环移位 s 位，如图 3-3 所示。

此外，每轮中使用的不同消息常量，其具体数值为：$K_t=\begin{cases}0x5a827999(0\leqslant t\leqslant 19)\\0x6ed9eba1(20\leqslant t\leqslant 39)\\0x8f1bbcdc(40\leqslant t\leqslant 59)\\0xca62c1d6(60\leqslant t\leqslant 79)\end{cases}$。

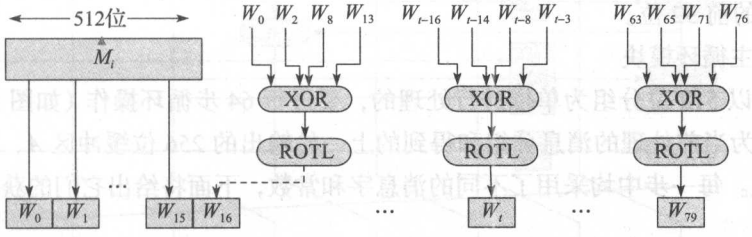


图 3-3　SHA-1 的 80 个消息字的产生过程

3.1.3　SHA-2 算法

2002 年，NIST 推出 SHA-2 系列 Hash 算法，其输出长度可取 224 位、256 位、384 位、512 位，分别对应 SHA-224、SHA-256、SHA-384、SHA-512。它还包含另外两个算法：SHA-512/224、SHA-512/256。SHA-2 系列 Hash 算法比之前的 Hash 算法具有更强的安全强度和更灵活的输出长度，其中 SHA-256 是常用的算法。下面将对前四种算法进行简单描述。

1. SHA-256 算法

SHA-256 算法的输入是最大长度小于 2^{64} 位的消息，输出是 256 位的消息摘要，输入消息以 512 位的分组为单位进行处理。算法描述如下。

（1）消息的填充

添加一个“1”和若干个“0”使其长度模 512 与 448 同余。在消息后附加 64 位的长度块，其值为填充前消息的长度。从而产生长度为 512 整数倍的消息分组，填充后消息的长度最多为 2^{64} 位。

（2）初始化链接变量

链接变量的中间结果和最终结果存储于 256 位的缓冲区中，缓冲区用 8 个 32 位的寄存器

A、B、C、D、E、F、G 和 H 表示，输出仍放在缓冲区以代替旧的 A、B、C、D、E、F、G、H。首先要对链接变量进行初始化，初始链接变量存储于 8 个寄存器 A、B、C、D、E、F、G 和 H 中：

$$A = H_0 = 0x6a09e667$$

$$B = H_1 = 0xbb67ae85$$

$$C = H_2 = 0x3c6ef372$$

$$D = H_3 = 0xa54ff53a$$

$$E = H_4 = 0x510e527f$$

$$F = H_5 = 0x9b05688c$$

$$G = H_6 = 0x1f83d9ab$$

$$H = H_7 = 0x5be0cd19$$

初始链接变量是取自前 8 个素数（2、3、5、7、11、13、17、19）的平方根的小数部分其二进制表示的前 32 位。

（3）处理主循环模块

消息块是以 512 位分组为单位进行处理的，要进行 64 步循环操作（如图 3-4 所示）。每一轮的输入均为当前处理的消息分组和得到的上一轮输出的 256 位缓冲区 A、B、C、D、E、F、G、H 的值。每一步中均采用了不同的消息字和常数，下面将给出它们的获取方法。

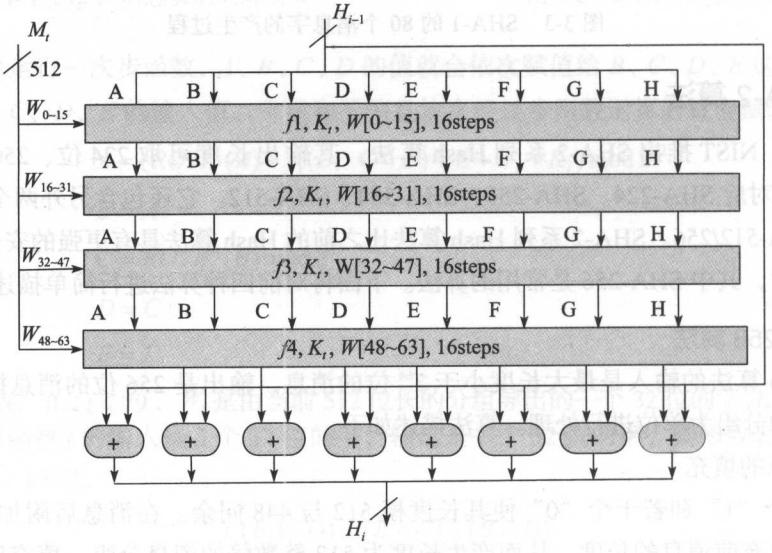


图 3-4 SHA-256 的压缩函数

（4）得出最终的 Hash 值

所有 512 位的消息块分组都处理完以后，最后一个分组处理后得到的结果即为最终输出

的 256 位的消息摘要。

步函数是 SHA-256 中最为重要的函数，也是 SHA-256 中最关键的部件。其运算过程如图 3-5 所示。

每一步都会生成两个临时变量，即 T_1、T_2：

$$T_1 = (\sum\nolimits_1(E) + Ch(E,F,G) + H + W_t + K_t) \bmod 2^{32}$$

$$T_2 = (\sum\nolimits_0(A) + Maj(A,B,C)) \bmod 2^{32}$$

根据 T_1、T_2 的值，对寄存器 A、E 进行更新。A、B、C、E、F、G 的输入值则依次赋值给 B、C、D、F、G、H。

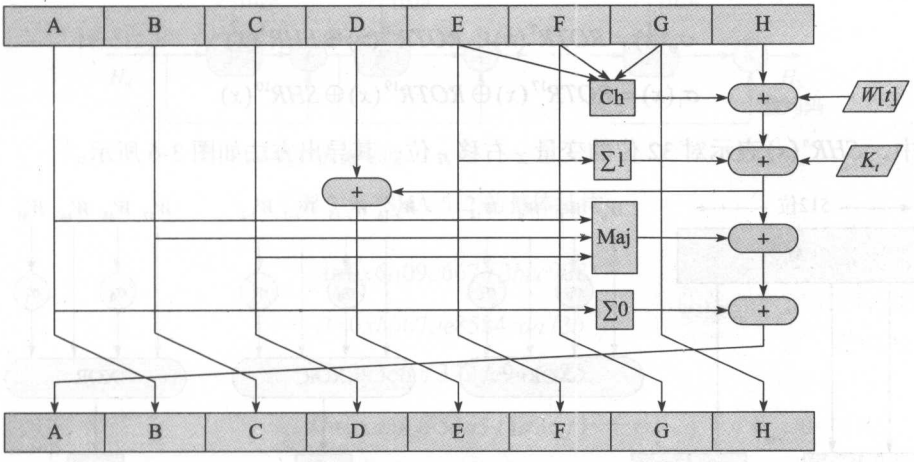


图 3-5　SHA-256 的步函数

$$A = (T_1 + T_2) \bmod 2^{32}$$

$$B = A$$

$$C = B$$

$$D = C$$

$$E = (D + T_1) \bmod 2^{32}$$

$$F = E$$

$$G = F$$

$$H = G$$

其中，　是步数，$0 \leqslant t \leqslant 63$；

$$Ch(E,F,G) = (E \wedge F) \oplus (\overline{E} \wedge G)\ ;$$

$$Maj(A,B,C) = (A \wedge B) \oplus (A \wedge C) \oplus (B \wedge C)\ ;$$

$$\sum\nolimits_0(A) = ROTR^2(A) \oplus ROTR^{13}(A) \oplus ROTR^{22}(A)\ ;$$

$$\sum_1(E) = ROTR^6(E) \oplus ROTR^{11}(E) \oplus ROTR^{25}(E)\ ;$$

且 $ROTR^n(x)$ 表示对 32 位的变量 x 循环右移 n 位。

K_t 的获取方法是取前 64 个素数（2，3，5，7，……）立方根的小数部分，将其转换为二进制，然后取这 64 个数的前 64 位作为 K_t。其作用是提供了 64 位随机串集合以消除输入数据里的任何规则性。

对于每个输入分组导出的消息分组 W_t，前 16 个消息字 $W_t(0 \leqslant t \leqslant 15)$ 直接按照消息输入分组对应的 16 个 32 位字，其他的则按照如下公式来计算得出：

$$W_t = W_{t-16} + \sigma_0(W_{t-15}) + W_{t-7} + \sigma_1(W_{t-2})\ ,\quad 16 \leqslant t \leqslant 63$$

其中：

$$\sigma_0(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)$$

$$\sigma_1(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$$

式中，$SHR^n(x)$ 表示对 32 位的变量 x 右移 n 位，其导出方法如图 3-6 所示。

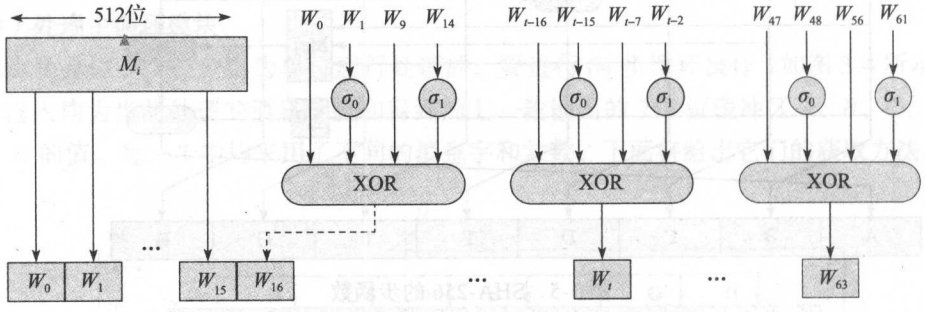


图 3-6　SHA-256 的 64 个消息字的生成过程

2. SHA-512 算法

SHA-512 是 SHA-2 中安全性能较高的算法，主要由明文填充、消息扩展函数变换和随机数变换等部分组成，初始值和中间计算结果由 8 个 64 位的移位寄存器组成。该算法允许输入的最大长度是 2^{128} 位，并产生一个 512 位的消息摘要，输入消息被分成若干个 1024 位的块进行处理，具体参数为：消息摘要长度为 512 位；消息长度小于 2^{128} 位；消息块大小为 1024 位；消息字大小为 64 位；步骤数为 80 步。图 3-7 显示了处理消息、输出消息摘要的整个过程，该过程的具体步骤如下。

1）消息填充：填充一个“1”和若干个“0”，使其长度模 1024 与 896 同余，填充位数为 0-1023，填充前消息的长度以一个 128 位的字段附加到填充消息的后面，其值为填充前消息的长度。

2）链接变量初始化：链接变量的中间结果和最终结果都存储于 512 位的缓冲区中，缓冲区用 8 个 64 位的寄存器 A、B、C、D、E、F、G、H 表示。初始链接变量也存储于 8 个寄

存器 A、B、C、D、E、F、G、H 中，其值为：

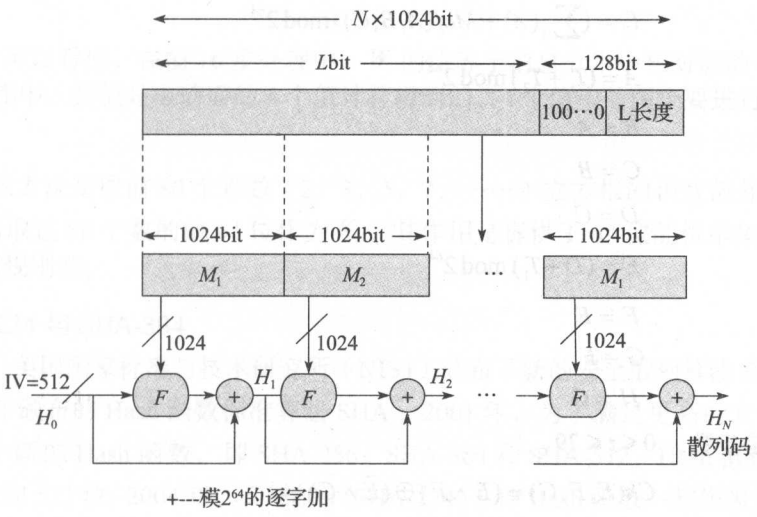


图 3-7　SHA-512 的整体结构

A=0x6a09e667f3bcc908

B=0xbb67ae8584caa73b

C=0x3c6ef372fe94f82b

D=0xa54ff53a5f1d36f1

E=0x510e527fade682d1

F=0x9b05688c2b3e6c1f

G=0x1f83d9abfb41bd6b

H=0x5be0cd19137e2179 。

初始链接变量采用 big-endian 方式存储，即字的最高有效字节存储于低地址位置。初始链接变量取自前 8 个素数的平方根的小数部分其二进制表示的前 64 位。

3）主循环操作：以 1024 位的分组为单位对消息进行处理，要进行 80 步循环操作。每一次迭代都把 512 位缓冲区的值 A、B、C、D、E、F、G、H 作为输入，其值取自上一次迭代压缩的计算结果，每一步计算中均采用了不同的消息字和常数。

4）计算最终的 Hash 值：消息的所有 N 个 1024 位的分组都处理完毕之后，第 N 次迭代压缩输出的 512 位链接变量即为最终的 Hash 值。

步函数是 SHA-512 中最关键的部件，其运算过程类似 SHA-256。每一步的计算方程如下所示，B、C、D、F、G、H 的更新值分别是 A、B、C、E、F、G 的输入状态值，同时生成两个临时变量用于更新 A、E 寄存器。

$$T_1 = (\sum_1(E) + Ch(E,F,G) + H + W_t + K_t) \bmod 2^{64}$$

$$T_2 = (\sum_0(A) + Maj(A,B,C)) \bmod 2^{64}$$

$$A = (T_1 + T_2) \bmod 2^{64}$$

$$B = A$$

$$C = B$$

$$D = C$$

$$E = (D + T_1) \bmod 2^{64}$$

$$F = E$$

$$G = F$$

$$H = G$$

其中，t 是步数，$0 \leqslant t \leqslant 79$。

$Ch(E,F,G) = (E \wedge F) \oplus (\overline{E} \wedge G)$；

$Maj(A,B,C) = (A \wedge B) \oplus (A \wedge C) \oplus (B \wedge C)$；

$\sum_0(A) = ROTR^{28}(A) \oplus ROTR^{34}(A) \oplus ROTR^{39}(A)$；

$\sum_1(E) = ROTR^{14}(E) \oplus ROTR^{18}(E) \oplus ROTR^{41}$。

对于 80 步操作中的每一步 t，使用一个 64 位的消息字 W_t，其值由当前被处理的 1024 位消息分组 M_i 导出，导出方法如图 3-8 所示。前 16 个消息字 $W_t(0 \leqslant t \leqslant 15)$ 分别对应消息输入分组之后的 16 个 32 位字，其他的则按照如下公式来计算得出：

$$W_t = W_{t-16} + \sigma_0(W_{t-15}) + W_{t-7} + \sigma_1(W_{t-2}), \quad 16 \leqslant t \leqslant 79$$

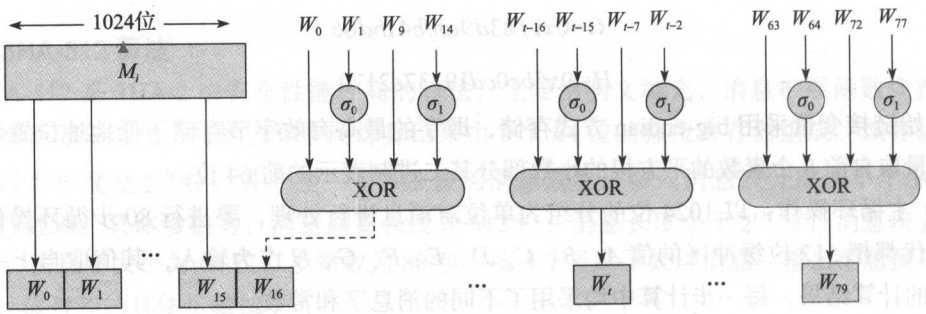


图 3-8 SHA-512 的 80 个消息字生成的过程

其中，

$$\sigma_0(x) = ROTR^1(x) \oplus ROTR^8(x) \oplus SHR^7(x)$$

$$\sigma_1(x) = ROTR^{19}(E) \oplus ROTR^{61}(x) \oplus SHR^6(x)$$

式中，$ROTR^n(x)$ 表示对 64 位的变量 x 循环右移 n 位，$SHR^n(x)$ 表示对 64 位的变量 x 右移 n 位。

从图 3-8 可以看出，在前 16 步处理中，W_t 的值等于消息分组中相对应的 64 位字，而余下的 64 步操作中，其值是由前面的 4 个值计算得到的，4 个值中的两个要进行移位和循环移位操作。

K_t 的获取方法是取前 80 个素数（2，3，5，7，……）立方根的小数部分，将其转换为二进制，然后取这 80 个数的前 64 位作为 K_t，其作用是提供了 64 位随机串集合以消除输入数据里的任何规则性。

3. SHA-224 与 SHA-384

1995 年，美国国家标准与技术研究所（NIST）公布了新的安全散列算法 SHA-1，该算法替代了 1993 年颁布的 Hash 函数标准算法 SHA；2001 年，为了满足更高的安全等级，NIST 又颁布了 3 个新的 Hash 函数，即 SHA-256、SHA-384 和 SHA-512，Hash 值的长度分别为 256 位、384 位和 512 位；2004 年，又增加了 SHA-224。5 种 Hash 函数一起作为安全散列标准。

SHA-256 和 SHA-512 是很新的 Hash 函数，前者定义一个字为 32 位，后者则定义一个字为 64 位。实际上二者的结构是相同的，只是在循环运行的次数、使用常数上有所差异。SHA-224 及 SHA-384 则是前述两种 Hash 函数的截短型，它们利用不同的初始值做计算。

SHA-224 的输入消息长度跟 SHA-256 的也相同，也是小于 2^{64} 位，其分组的大小也是 512 位，其处理流程跟 SHA-256 也基本一致，但是存在如下两个不同的地方。

- SHA-224 的消息摘要取自 A、B、C、D、E、F、G 共 7 个寄存器的比特字，而 SHA-256 的消息摘要取自 A、B、C、D、E、F、G、H 共 8 个寄存器的 32 比特字。
- SHA-224 的初始链接变量与 SHA-256 的初始链接变量不同，它采用高端格式存储，但其初始链接变量的获取方法是取前第 9 至 16 个素数（23、29、31、37、41、43、47、53）的平方根的小数部分其二进制表示的第二个 32 位，SHA-224 的初始链接变量如下：

$$A = 0xc1059ed8$$

$$B = 0x367cd507$$

$$C = 0x3070dd17$$

$$D = 0xf70e5939$$

$$E = 0xffc00b31$$

$$F = 0x68581511$$

$$G = 0x64f98fa7$$

$$H = 0xbefa4fa4$$

SHA-224 的详细计算步骤与 SHA-256 一致。

SHA-384 的输入消息长度跟 SHA-512 相同，也是小于 2^{128} 位，而且其分组的大小也是

1024 位，处理流程跟 SHA-512 也基本一致，但是也有如下两处不同的地方。

- SHA-384 的 384 位的消息摘要取自 A、B、C、D、E、F 共 6 个 64 比特字，而 SHA-512 的消息摘要取自 A、B、C、D、E、F、G、H 共 8 个 64 比特字。
- SHA-384 的初始链接变量与 SHA-512 的初始链接变量不同，它也采用高端格式存储，但其初始链接变量的获取方法是取前 9 至 16 个素数（23、29、31、37、41、43、47、53）的平方根的小数部分其二进制表示的前 64 位，SHA-384 的初始链接变量如下：

$$A = 0xcbbb9d5dc1059ed8$$

$$B = 0x629a292a367cd507$$

$$C = 0x9159015a3070dd17$$

$$D = 0x152fecd8f70e5939$$

$$E = 0x67332667ffc00b31$$

$$F = 0x8eb44a8768581511$$

$$G = 0xdb0c2e0d64f98fa7$$

$$H = 0x47b5481dbefa4fa4$$

SHA-384 的详细计算步骤与 SHA-512 的相同。

3.1.4 SHA-3 算法

由于 MD5、SHA 系列的 Hash 函数遭受到了碰撞攻击，NIST 在 2005 年 10 月 31 日到 11 月 1 日和 2006 年 8 月 24 日至 25 日举办了两次 Hash 函数研讨会，评估了 Hash 函数当前的使用状况，征求了公众对 Hash 函数的新准则。经过讨论之后，在 2007 年 11 月，NIST 决定通过公开竞赛，以高级加密标准 AES 的开发过程为范例开发新的 Hash 函数，新的 Hash 算法被称为 SHA-3，用于扩充包含 SHA-2 算法在内的 FIPS 180-3 中的安全 Hash 标准。截止到 2008 年 10 月 31 日，有 64 个算法提交到 NIST。2008 年 12 月 10 日，NIST 宣布在这 64 个算法中有 51 个算法满足对候选算法提出的可接受的最低标准，确定为第一轮的候选，并开始进行 SHA-3 的第一轮竞选。提交的第一轮候选算法公布在 www.nist.gov/hash-competition 上以用于公众的评审。到 2009 年 7 月 24 日，第一轮的候选算法中剩下 14 个候选算法进入到第二轮的竞选中，这 14 个第二轮的候选算法为 BLAKE、BLUE MIDNIGHT WISH、CubeHash、ECHO、Fugue、GrØstl、Hamsi、JH、Keccak、Luffa、Shabal、SHAvite-3、SIMD 和 Skein。

NIST 于 2010 年 8 月 23 日至 24 日在加州大学圣塔芭芭拉分校举行第二次 SHA-3 候选会议。在 2010 年 12 月 9 日，NIST 宣布 5 个候选算法进入到第三轮，也是最后一轮的竞选，这 5 个候选分别是：BLAKE、GrØstl、JH、Keccak 和 Skein。其中，BLACK 使用 HAIFA 迭代框架，在压缩函数中加入盐和计数器，内部结构是局部宽管道结构；GrØstl 和 JH 采用宽管道 MD 结构，能抵抗一般的通用攻击；Keccak 使用 Sponge 函数；Skein 既可以使用迭代结

构，也可以使用树结构。NIST 在 2012 年评选出最终算法并产生了新的 Hash 标准。Keccak 算法由于其较强的安全性和软硬件实现性能，最终被选为新一代的标准 Hash 算法，并被命名为 SHA-3。

SHA-3 算法整体采用 Sponge 结构，分为吸收和榨取两个阶段。SHA-3 的核心置换 f 作用在 $5\times5\times64$ 的三维矩阵上。整个 f 共有 24 轮，每轮包括 5 个环节 θ、ρ、π、χ、τ。算法的 5 个环节分别作用于三维矩阵的不同维度之上。θ 环节是作用在列上的线性运算；ρ 环节是作用在每一道上的线性运算，将每一道上的 64 比特进行循环移位操作；π 环节是将每道上的元素整体移到另一道上的线性运算；χ 环节是作用在每一行上的非线性运算，相当于将每一行上的 5 比特替换为另一个 5 比特；τ 环节是加常数环节。

目前，公开文献对 SHA-3 算法的安全性分析主要是从以下几个方面来展开的。

- 对 SHA-3 算法的碰撞攻击、原像攻击和第二原像攻击。
- 对 SHA-3 算法核心置换的分析，这类分析主要针对算法置换与随机置换的区分来展开。
- 对 SHA-3 算法的差分特性进行展开，主要研究的是 SHA-3 置换的高概率差分链，并构筑差分区分器。

Keccak 算法的立体加密思想和海绵结构，使 SHA-3 优于 SHA-2，甚至 AES。Sponge 函数可建立从任意长度输入到任意长度输出的映射。

3.1.5　RIPEMD160 算法

RIPEMD（RACE Integrity Primitives Evaluation Message Digest），即 RACE 原始完整性校验消息摘要，是比利时鲁汶大学 COSIC 研究小组开发的 Hash 函数算法。RIPEMD 使用 MD4 的设计原理，并针对 MD4 的算法缺陷进行改进，1996 年首次发布 RIPEMD-128 版本，它在性能上与 SHA-1 相类似。

RIPEMD-160 是对 RIPEMD-128 的改进，并且是 RIPEMD 中最常见的版本。RIPEMD-160 输出 160 位的 Hash 值，对 160 位 Hash 函数的暴力碰撞搜索攻击需要 2^{80} 次计算，其计算强度大大提高。RIPEMD-160 的设计充分吸取了 MD4、MD5、RIPEMD-128 的一些性能，使其具有更好的抗强碰撞能力。它旨在替代 128 位 Hash 函数 MD4、MD5 和 RIPEMD。

RIPEMD-160 使用 160 位的缓存区来存放算法的中间结果和最终的 Hash 值。这个缓存区由 5 个 32 位的寄存器 A、B、C、D、E 构成。寄存器的初始值如下所示：

$$A = 67452301$$

$$B = efcdab89$$

$$C = 98badcfe$$

$$D = 10325476$$

$$E = c3d2e1f0$$

数据存储时采用低位字节存放在低地址上的形式。

处理算法的核心是一个有 10 个循环的压缩函数模块，其中每个循环由 16 个处理步骤组成。在每个循环中使用不同的原始逻辑函数，算法的处理分为两种不同的情况，在这两种情况下，分别以相反的顺序使用 5 个原始逻辑函数。每一个循环都以当前分组的消息字和 160 位的缓存值 A、B、C、D、E 为输入得到新的值。每个循环使用一个额外的常数 K，在最后一个循环结束后，两种情况的计算结果 A、B、C、D、E 和 A'、B'、C'、D'、E' 及链接变量的初始值经过一次相加运算产生最终的输出。对所有的 512 位的分组处理完成之后，最终产生的 160 位输出即为消息摘要。

除了 128 位和 160 位的版本之外，RIPEMD 算法也存在 256 位和 320 位的版本，它们共同构成 RIPEMD 家族的四个成员：RIPEMD-128、RIPEMD-160、RIPEMD-256、RIPEMD-320。其中 128 位版本的安全性已经受到质疑，256 位和 320 位版本减少了意外碰撞的可能性，但是相比于 RIPEMD-128 和 RIPEMD-160，它们不具有较高水平的安全性，因为他们只是在 128 位和 160 位的基础上，修改了初始参数和 s-box 来达到输出为 256 位和 320 位的目的。

3.2 椭圆曲线密码

由于 Internet 网在全球范围内的迅速流行和普及，使得通过网络传输各种信息和数据的交换量迅猛增加，所以针对网络信息的安全问题日益突显出来，这也使得对于各类信息的加密研究显得尤为重要。为了保证网络传输中各类信息的安全，目前通常使用的信息加密技术根据密钥类型不同可以划分为对称加密系统和非对称加密系统两大类。当前，普遍认为比较安全有效的公钥密码系统主要包括 RSA 体制、ElGamal 体制和椭圆曲线密码体制等。椭圆曲线密码体制（Elliptic Curve Cryptosystem，ECC）是 1985 年由 Koblitz N 和 Miller V 提出的，其安全性是建立在求解椭圆曲线离散对数问题困难性的基础上的，在同等密钥长度的情况下 ECC 的安全强度要远高于 RSA 体制等其他密码体制，因而 ECC 在网络信息安全领域有着非常重要的理论研究价值和广阔的实际应用前景。另一方面，在安全性相当的情况下，ECC 所使用的密钥长度更短，这也就意味着对于带宽和存储空间的需求相对较小，并且到目前为止，还没有出现针对椭圆曲线的亚指数时间算法。因此，ECC 将会是今后最重要的主流公钥加密技术。

椭圆曲线密码体制的安全性，依赖于椭圆曲线上离散对数问题（Elliptic Curve Discrete Logarithm Problem，ECDLP）的难解性。而对椭圆曲线密码体制的攻击，也可以归结到对 ECDLP 的攻击。对 ECDLP 的攻击类似于对有限域的乘法群上离散对数问题的攻击，但是攻击方法并不能有效地移植到 ECDLP。对 ECDLP 的攻击主要分为两类：一类是对所有曲线的离散对数问题的攻击；另一类是对特殊曲线的离散对数问题的攻击。

目前，椭圆曲线公钥密码体系开始从学术理论研究阶段走向实际应用阶段，受到学术界、开发商、政府部门、密码标准研究组织等有关各界的重视，IEEE、ANSI、ISO、IETF 等组织已在椭圆曲线密码算法的标准化方面做了大量的工作。本节将简单介绍椭圆曲线的定

义、相关理论及公私钥的产生方法。

3.2.1　椭圆曲线方程

1. 椭圆曲线的定义

椭圆曲线是由一个具有两个变元 x 和 y 的魏尔斯特拉斯方程：

$$y^2 + axy + by = x^3 + cx^2 + dx + e$$

所确定的所有点 (x,y) 组成的集合，外加一个无穷远点 O（认为其 y 坐标无穷大）。

常用于密码系统的基于有限域 $GF(p)$ 上的椭圆曲线是由方程：

$$y^2 = x^3 + ax + b(\mathrm{mod}\ p)$$

所确定的所有点 (x,y) 组成的集合，外加一个无穷远点 O。其中 a、b、x、y 均在 $GF(p)$ 上取值，且有 $4a^3 + 27b^2 \neq 0$，p 是大于 3 的素数，通常用 $Ep(a,b)$ 来表示这类曲线。

2. 椭圆曲线在模 P 下的 Abel 群

定义 $E_p(a,b)$ 为在模 p 之下椭圆曲线 E 上所有的整数点构成的集合（包括 O）。椭圆曲线上的点集合 $E_p(a,b)$ 可根据如下定义的加法规则构成一个 Abel 群。

如果椭圆曲线上的 3 个点位于同一条直线上，那么它们的和为 O。进一步可按如下形式定义椭圆曲线上的加法法则，具体如下。

1）$O+O=O$。

2）O 为加法单位元，即对椭圆曲线上的任一点 P，有 $P+O=P$。因为 P 与 $-P$ 在曲线上的第三个交点是 O。

3）设 $P_1=(x,y)$ 是椭圆曲线上的一点，它的加法逆元定义为 $P_2=-P_1=(x,-y)$。原因是椭圆曲线上的 3 点 P_1、P_2、O 共线，所以 $P_1+P_2+O=\mathrm{O}, P_1+P_2=\mathrm{O}$，即 $P_2=-P_1$。由 $O+O=O$，还可得 $O=-O$。

4）设 Q 和 R 是椭圆曲线上 x 坐标不同的两点，$Q+R$ 的定义如下：画一条通过 Q、R 的直线与椭圆曲线交于 P_1，由 $Q+R+P_1=O$ 可得 $Q+R=-P_1$。

5）对于所有的点 P 和 Q，满足加法交换律，即 $P+Q=Q+P$。

6）对于所有的点 P、Q 和 R，满足加法结合律，即 $P+(Q+R)=(P+Q)+R$。

7）点 Q 的倍数定义如下：在 Q 点做椭圆曲线的一条切线，设切线与椭圆曲线交于点 S，定义 $2Q=Q+Q=-S$。类似地可定义 $3Q=Q+Q+Q$。

8）设 $P=(x_1,y_1)$，$Q=(x_2,y_2)$，$P\neq -Q$，则 $P+Q=(x_3,y_3)$ 由以下规则可确定：

$$x_3=\lambda^2-x_1-x_2(\mathrm{mod}\ p)\ ,\quad y_3=\lambda(x_1-x_3)-y_1(\mathrm{mod}\ p)\ ,$$

$$\lambda=\begin{cases}\dfrac{y_2-y_1}{x_2-x_1}, P\neq Q\\ \dfrac{3x_1^2+a}{2y_1}, P=Q\end{cases}$$

以上规则的几何意义具体如下。

1）O 是加法单位元。

2）一条与 x 轴垂直的线和曲线相交于两个点，这两个点的 x 坐标相同，即 $P_1=(x,y)$ 和 $P_2=(x,-y)$，同时它也与曲线相交于无穷远点 O，因此 $P_2=-P_1$，故椭圆曲线的性质决定了 P 与其逆元成对出现在椭圆曲线上。

3）横坐标不同的两个点 P 和 Q 相加时，先在它们之间画一条直线并求直线与曲线的第三个交点 R，则 $P+Q=-R$。

4）两个相同的点 P 相加时，通过该点画一条切线，切线与曲线交于另一个点 R，则 $P+P=2P=-P$。

椭圆曲线点乘规则具体如下。

1）如果 k 为整数，则对所有的点 $P\in E_p(a,b)$，有：

$$kP=P+P+...+P \quad (k\text{ 个 }P\text{ 相加})$$

2）如果 s 和 t 为整数，则对所有的点 $P\in E_p(a,b)$，有：

$$(s+t)P=sP+tP,\quad s(tP)=(st)P$$

若存在最小正整数 n，使得 $nP=O(P\in E_p(a,b))$，则 n 为椭圆曲线 E 上点 P 的阶（n 是椭圆曲线的阶 N 的因子）。除了无限远的点 O 之外，椭圆曲线 E 上任何可以生成所有点的点都可以视为 E 的生成元，但并不是所有在 E 上的点都可视为生成元。

3.2.2 公钥和私钥的产生算法

椭圆曲线上离散对数的定义如下：给定素数 P 和椭圆曲线 E，对 $Q=kP$，在已知 P、Q 的情况下求出小于 P 的正整数 k。可以证明，已知 k 和 P 计算 Q 比较容易，而由 Q 和 P 计算 k 则比较困难，至今还没有有效的方法来解决这个问题，这就是椭圆曲线加密算法原理之所在。例如，有限域 GF(23) 上的椭圆曲线 $y^2=x^3+ax+b$，求 $Q=(x_1,y_1)$ 对于 $P=(x,y)$ 的离散对数。最直接的方法就是计算 P 的倍数，$P=(x,y)$，$2P=(x_2,y_2)$，$3P=(x_3,y_3)$，…，直到找到 k 使得 $kP=(x_1,y_1)=Q$。因此，Q 关于 P 的离散对数是 k。然而，对于大素数构成的群 E，这样计算离散对数是不现实的。事实上，目前还不存在多项式时间算法求解椭圆曲线上的离散对数问题，所以通常假设此类问题是困难问题。

基于上述椭圆曲线上的离散对数难题，下面考虑如何生成用户的公私钥对，其步骤具体如下。

1）选择一个椭圆曲线 E：$y^2\equiv x^3+ax+b(\mathrm{mod}\,p)$，构造一个椭圆曲线 Abel 群 $E_p(a,b)$。

2）在 $E_p(a,b)$ 中挑选生成元点 $G=(x_0,y_0)$，G 应使得满足 $nG=O$ 的最小 n 是一个非常大的素数。

3）选择一个小于 n 的整数 n_B 作为其私钥，然后产生其公钥 $P_B=n_BG$，则用户的公钥为 (E,n,G,P_B)，私钥为 n_B。

下面来看一个计算示例。

以 $E_{23}(1,1)$ 为例，设 G = (3, 10)，若选择私钥 $d=2$，计算公钥 $Q=2G$。

计算过程如下：

$$\lambda=(3\times3^2+1)/(2\times10)=5/20=1\times4^{-1}\equiv1\times6=6 \bmod 23$$

$$x_3=6^2-3-3=30\equiv7 \bmod 23$$

$$y_3=6\times(3-7)-10=-34\equiv12 \bmod 23$$

所以，公钥 $Q=2G=(7,12)$ 仍为 $E_{23}(1,1)$ 上的点。

椭圆曲线密码体制是现有公钥密码体制中比特位强度最高的密码体制，其发展前景相当广泛，且能够适应当代社会的需求。但就目前而言，还面临着很多理论及技术上的问题，比如，如何选取合适的有限域 $GF(Q^m)$ 的椭圆曲线 E，如何选取基点 P 对于 ECC 的速度、效率、密钥长度及安全性等，这些都是至关重要的。

3.3　ECDSA 数字签名

与普通的离散对数问题（Discrete Logarithm Problem，DLP）和大数分解问题（Integer Factorization Problem，IFP）不同，椭圆曲线上的离散对数问题 ECDLP 没有亚指数时间的解决方法。因此椭圆曲线密码的单位比特强度要高于其他公钥体制。

椭圆曲线数字签名算法（Elliptic Curve Digital Signature Algorithm，ECDSA）基于 ECDLP，是使用椭圆曲线对数字签名算法（Digital Signature Algorithm，DSA）的模拟。ECDSA 首先由 Scott 和 Vanstone 在 1992 年为了响应 NIST 对数字签名标准（DSS）的要求而提出的。ECDSA 于 1998 年作为 ISO 标准被采纳，在 1999 年作为 ANSI 标准被采纳，并于 2000 年成为 IEEE 和 FIPS 标准。

下面来看看 ECDSA 数字签名算法。

该算法密钥的生成方式为：选取一条椭圆曲线 $E_p(a,b)$。取 $E_p(a,b)$ 的一个生成元 G，$E_p(a,b)$ 和 G 作为公开参数。如 3.2.2 节所述，用户 A 随机选取 d 作为私钥，并计算 $Q=dG$ 作为公钥。

签名的产生方式为：签名者 A 利用私钥对消息 m 进行签名，具体方法如下。

1）随机选择一个整数 k。

2）计算 $kG=(x_1,y_1)$，$r=x_1 \bmod n$。

3）计算 $e=H(m)$，$s=k^{-1}(e+dr) \bmod n$。

其中 (r,s) 是 A 对消息 m 的签名。

ECDSA 数字签名算法的签名验证方式为：验证者 B 验证 (r,s) 是否 A 对消息 m 的签名，具体方法如下。

1）验证 r,s 是 [1, n−1] 中的整数。

2）计算 $e=H(m)$，$w=s^{-1} \bmod n$，$u_1=ew \bmod n$，$u_2=rw \bmod n$。

3）计算 $X=u_1G+u_2Q=(x_1,y_1)$。

当且仅当 $x_1 \bmod n=r$ 时接受这个签名。

ECDSA 在安全性方面的目标是能抵抗选择明文（密文）攻击。而攻击 A 的攻击者目标是在截获 A 的签名之后，可以生成对任何消息的合法签名。尽管 ECDSA 的理论模型很坚固，但是人们仍然研究出了很多措施以提高 ECDSA 的安全性。在 ECDLP 不可破解及 Hash 函数足够强大的前提下，DSA 和 ECDSA 的一些变形已被证明可以抵抗现有的任何选择明文（密文）攻击。在椭圆曲线所在群是一般群，并且 Hash 函数能够抗碰撞攻击的前提下，ECDSA 本身的安全性已经得到了证明。

比特币区块链中，每个交易都需要用户使用私钥签名，只有采用该用户公钥验证通过的交易，比特币网络才会承认。

3.4 Schnorr 数字签名

3.4.1 技术思想

基于 ElGamal 签名与离散对数问题，1989 年 Schnorr 提出了一种随机化的签名方案，称为 Schnorr 数字签名方案。下面来具体看一下。

设 p 和 q 是满足 $p \equiv 1 \bmod q$ 的两个素数，一般取 $p \approx 2^{1024}$，$q=2^{160}$，此时 p 为 1024 位的数，q 为 160 位的数，其中 160 位正是 SHA-1 的 Hash 值长度。Schnorr 签名体制对 ElGamal 签名方案以独特的方式进行了修改，使得长度为 $\log_2 q$ 位的消息签名包含长度为 $2\log_2 q$ 位的签名，但是它的计算是在 Z_p 上进行的。它是通过工作在 Z_p^* 上的 q 元子群来实现的。该方案的安全性是基于这样的思想来设计的：在特定的 Z_p^* 子群上求解离散对数是安全的。Schnorr 签名中的密钥与 ElGamal 签名方案中的相似。

Schnorr 数字签名方案的具体描述如下。

- 密钥生成：用户随机选择 x 作为私钥，并计算 $y=g^x \bmod p$ 作为公钥。
- 签名产生：签名者选择随机数 $0<k<q$，计算 $r=g^k \bmod p$，$e=H(r \| m)$，$s=k+xe \bmod q$。(e,s) 是用户 A 对消息 m 的签名。
- 签名验证：验证者收到消息 m 与签名 (e,s) 后，计算 $r'=g^s y^{-e} \bmod p$，$e'=H(r' \| m)$。若 $e'=e \bmod p$，则接受签名，否则拒绝。

事实上，$r'=g^s y^{-e} \bmod p=g^s (g^x)^{-e} \bmod p=g^{s-xe} \bmod p=g^k \bmod p$，从而有 $H(r \| m)=H(r' \| m)$，即 $e'=e \bmod p$。

3.4.2 Schnorr 与 ECDSA 的异同

Schnorr 数字签名算法基于离散对数困难性问题，可扩展至椭圆曲线上的 Schnorr 签名算

法，优势表现为：可实现多重数字签名与批验证，即多个用户对同一消息的签名可聚合成单个，且仅需要单次验证过程。相较于 ECDSA，其具有更短的签名、更强的安全性、更快的签名 / 验证时间。

3.5 Bloom filter

Bloom filter 是一种节省空间、高效率的数据表示和查询结构。它可以利用位数组很简洁地表示一个集合，并能以很高的概率判断一个元素是否属于这个集合。因此，这种数据结构适合应用在能容忍低错误率的场合。

Bloom filter 是由 Howard Bloom 在 1970 年提出的二进制向量数据结构，用于解决某些元素是否为集合中元素的判断问题。它突破了传统 Hash 函数的映射和存储元素的方式，通过一定的错误率换取了空间的节省和查询的高效。二十世纪八十年代，随着 PC 应用的推广，Bloom filter 的应用开始推广，比如应用于高效地解决拼写检查问题，解决多处理器计算机中数据库的连接问题等。网络时代的到来使得 Bloom filter 的应用越来越多，如应用到分布式数据库中进行查询，应用到网络中取代 ICP 以进行高速缓存查询，应用到 P2P 中进行高效的联合查询等。

3.5.1 技术原理

如果需要判断一个元素是不是在一个集合中，通常的做法是把所有元素都保存下来，然后通过比较确定它是不是在集合之内，链表、树都是基于这种思路，当集合内的元素个数增大时，我们需要的空间和时间都会线性变大，检索速度也会越来越慢。Bloom filter 采用的是 Hash 函数的方法，将一个元素映射到一个 m 长度的阵列上的一个点，当这个点是1时，那么这个元素在集合内，反之则不在集合内。这个方法的缺点就是当检测的元素很多的时候可能会有冲突，解决方法就是使用 k 个 Hash 函数对应 k 个点，如果所有点都是1的话，那么元素在集合内，如果有 0 的话，那么元素不在集合内。

工作原理叙述如下。

1）初始状态时，Bloom filter 是一个包含 m 位的位数组，每一位都置为 0，如图 3-9 所示。

0	0	0	0	0	0	0	0	0	0	0	0

图 3-9　Bloom filter 初始状态

2）为了表达 $S=\{x_1,x_2,...,x_n\}$ 这样有 n 个元素的集合，Bloom filter 使用 k 个相互独立的 Hash 函数，它们分别将集合中的每个元素映射到 $\{1,2,...,m\}$ 的范围中。对于任意一个元素 x，第 i 个 Hash 函数映射的位置 $h_i(x)$ 都会被置为 1，如图 3-10 所示。

3）在判断 y 是否属于这个集合时，可对 y 应用 k 次 Hash 函数，如果所有 $h_i(y)$ 的位置都是 $1(1\leqslant i\leqslant k)$，那么就认为 y 是集合中的元素，否则就认为 y 不是集合中的元素。例如，在图 3-11 中，y_1 就不是集合中的元素，y_2 或者属于这个集合，或者刚好是一个误识。

Bloom filter 的优点在于它的插入和查询时间都是常数，另外它查询元素却不保存元素本

身，具有良好的安全性。它的缺点也是显而易见的，当插入的元素越多，错判“在集合内”的概率就越大了，另外 Bloom filter 也不能删除任何一个元素，因为多个元素散列的结果可能在 Bloom filter 结构中占用的是同一个位，如果删除了一个比特位，可能会影响多个元素的检测。

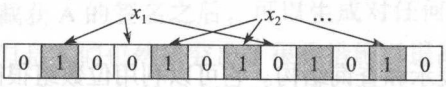


图 3-10 Bloom filter 映射图示

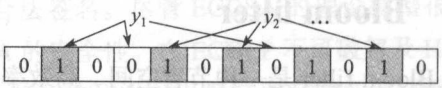


图 3-11 Bloom filter 判断图示

3.5.2 应用案例

Bloom filter 的应用包括：表示一种压缩数据的集合；替代原始的数据集合；完成元素是否在集合中的查询判断，如数据库操作、字典查询和文件操作方面等；Bloom filter 也广泛应用于网络领域，包括 P2P 网络、资源路由、数据帧路由标签、网络测量管理、网络入侵检测、传感器网络数据过滤和路由等；第三类则是元素表示的保密性。

下面就来详细介绍一些应用案例。

1. 在网络爬虫中的应用

Bloom filter 的优劣主要与 Hash 函数的质量相关，而且 Hash 函数之间的相关度越小越好，每个 Hash 函数本身的计算过程不要太复杂，不然会影响效率。理想情况下是取 k 个完全不相关的 Hash 函数，在不是很严格情况下，也可以通过一个 Hash 函数的参数变化产生 k 个不同的 Hash 函数，比如将 i（$1 \leqslant i \leqslant k$）作为参数参与 Hash 函数的计算。不同的应用场景下，Hash 函数的设计方法也不相同。在网络爬虫的设计中，一般采用 MD5 算法来构造 Hash 函数。把要过滤的地址散列到一片很大的位地址空间。Bloom filter 类实现了基于 Bloom filter 的 URL 消重算法，将 GetUrl 类中提取的 URL 地址进行消重。

2. 加速查询

适用于一些 key-value 存储系统，当 values 存储在硬盘时，查询就是一件很费时的事。将 Storage 的数据都插入 Filter，若在 Filter 中查询都不存在时，那就不需要去 Storage 中查询了。当 False Position 出现时，只是会导致一次多余的 Storage 查询。

Google 的 BigTable 也使用了 Bloom filter，以减少不存在的行或列在磁盘上的查询，大大提高了数据库查询操作的性能。在 Internet Cache Protocol 中的 Proxy-Cache 很多都是使用 Bloom filter 存储 URLs 的。Bloom filter 除了具有高效的查询功能之外，还能很方便地传输交换 Cache 信息。

3. 重复数据删除中的应用

主流的重复数据删除技术的基本原理是对文件进行定长或变长分块，然后利用 Hash 函数计算数据块指纹，如果两个数据块指纹相同则认为是重复数据块（同样这里存在数据碰撞

的问题），只保存一个数据块副本即可，其他相同的数据块使用该副本索引号表示，从而实现数据缩减存储空间并提高存储效率。为了查询一个数据块是否重复或是否已经存在，需要计算数据块指纹并进行查找，并记录所有唯一数据块的指纹。

引入 Bloom filter 机制后，对于一个新数据块，首先查找 Bloom filter，如果未命中则说明这是一个新的唯一数据块，直接保存数据块合并 Cache 数据块信息即可；如果命中，则说明这有可能是一个重复数据块，需要通过进一步的 Hash 或 tree 查找进行确认，此时需要 Cache 与 Disk 进行交互。受益于 Bloom filter 及 Cache，DataDomain 系统可以减少 99% 的磁盘访问，从而可以利用少量的内存空间大幅提高数据块查重的性能。

Chapter 4 第 4 章

比特币区块链开发

本章会以一个程序开发者的视角对比特币相关的代码进行剖析，从而让读者可以自行编译、阅读、修改、运行相关代码，以及开发第三方应用。本章假定受众是一位了解 C++ 语言的程序开发者，且已经深刻理解了比特币的相关概念，包括挖矿、交易、区块链、Hash、公私钥、签名等，因此，除非特别必要，否则将会略过相应的概念解释。

4.1 Bitcoin 的编译过程

Linux、Mac、Windows 等平台均支持编译运行比特币代码，作为开发人员，可以重点研究其中的 bitcoind 代码部分，因为这是比特币协议及区块链的核心，至于面向终端用户的图形界面（GUI）部分，完全可以放心地忽略，对后续研究毫无影响。

4.1.1 Ubuntu 下的编译

在 Linux 下可选择 Ubuntu 14.04 版本作为开发测试环境，这也是目前在阿里云和 linode 等云服务器中部署产品时常用的版本。

1. 环境准备

通过以下命令可安装并编译 bitcoind 所需要的依赖库：

```
sudo apt-get install build-essential libtool autotools-dev autoconf automake
libssl-dev libboost-all-dev libdb-dev libdb++-dev pkg-config libevent-devgit-core
```

2. 复制 Bitcoin 源代码并进入其目录

通过以下命令可复制 Bitcoin 源代码，并且进入它的目录：

```
git clone https://github.com/bitcoin/bitcoin
cd bitcoin
```

3. 编译 bitcoind

首先，生成编译源码所需要的库配置，命令如下：

```
./autogen.sh
```

然后，生成 makefile 文件，命令如下：

```
./configure--without-gui
```

如果在生成 makefile 文件时，遇到这样的提示：

```
configure: error: Found Berkeley DB other than 4.8, required for portable wallets
```

那就可以使用如下命令：

```
./configure--without-gui —with-incompatible-bdb
```

若禁用钱包界面功能，仅提供比特币网络节点功能，则使用如下命令：

```
./configure --without-gui—disable-wallet
```

接下来，就是利用 make 进行编译了，命令如下：

```
make -j
```

编译好的 bitcoind、bitcoin-tx 和 bitcoin-cli 在 src 目录下。

最后，安装编译好的二进制文件（可选），命令如下：

```
make install
```

4.1.2　Mac 下的编译

由于 Mac 用的是 Unix 内核，因此其编译过程与 Ubuntu 基本类似，最大的不同之处是环境的准备过程，可以重点关注这一部分。

1. 环境准备

在进行环境准备时，要先安装好 Xcode 和 homebrew http://brew.sh/。然后，在 Mac 的 shell 下用 brew 安装依赖，命令如下：

```
brew install autoconf automake berkeley-db4 libtool boost miniupnpc openssl
pkg-config protobuf
```

2. 复制 Bitcoin 源代码并进入其目录

通过以下命令可复制 Bitcoin 源代码，并且进入它的目录：

```
git clone https://github.com/bitcoin/bitcoin.git
cd bitcoin
```

3. 编译 bitcoind

编译命令如下：

```
./autogen.sh
./configure--without-gui
make
```

编译好的 bitcoind、bitcoin-tx 和 bitcoin-cli 在 src 目录下。

4.1.3 Windows 下的编译

Windows 下的环境准备过程相对比较复杂，若手动安装，则需要下载安装形形色色的依赖包；而在类 Unix 系统下，只需要使用 apt-get 或 brew 命令即可，所以建议比特币开发环境尽量选择 Linux 或 Mac 系统。

不过，有一个好消息，最近有一个名为 phelix 的开发者制作了一个开源的批处理脚本 EasyWinBuilder，基本实现了编译的半自动化，大大提高了安装效率，更方便的是通过修改 set_vars.bat 里的配置参数，可以编译从比特币克隆出来的其他各种竞争币。使用该批处理脚本的方法具体如下：

从 https://github.com/phelix/easywinbuilder 下载 EasyWinBuilder 到一个不包含中文的目录中，如 c:\ 盘，解压缩进入项目目录后，双击 __all_easywinbuilder.bat，跟随命令行提示会依次自动安装 MinGW/MSYS 编译环境、第三方依赖包，以及自动编译比特币执行文件。

不过，在安装时可能会碰到如下几个问题。

- 首先，在 Windows XP 系统下有可能存在找不到 mkdir 等的各种问题，因此，一定要在 Windows 7 以上的环境中安装。
- 另外，在 Windows 下安装时，需要从 Github 上下载某些包，如 libevent、protobuf 等，这些包目前部署在亚马逊 AWS 上，但是亚马逊 AWS 在国内是无法打开的，所以需要提前自行配置一下 VPN。
- 此外，qtbase-opensource-src-5.3.2.zip 用 unzip 自动解压时可能会失败，若出现该问题，请手工解压到 Qt/5.3.2 目录下。
- 最后，批处理中某些 autogen.sh 可能会被卡住。此时可以删除该 bat 里的 autogen.sh，然后把 EasyWinBuilder 项目目录下解压出来的 mingw32/bin 目录全路径加入环境变量 PATH 中，最后进入 msys 环境手动执行 autogen.sh。

不要爱上编译，不要浪费时间折腾编译过程，找一台 Mac，下载代码，熟悉框架，理解代码，快速开发才是正途。

下面就来分析代码。

4.2　代码剖析

这里选取 Bitcoin 目前的最新版本 v0.12.1 作为分析基础。注意此书只分析几大核心模块的核心流程及数据结构，至于细枝末节就不展开来讲了，明白了主要脉络，再按图索骥即可读懂代码。

如果您还没有按照 4.1 节的编译过程复制 Bitcoin 的源代码，那么也可以简单地执行如下命令：

```
git clone https://github.com/bitcoin/bitcoin
cd bitcoin
```

然后，在 Bitcoin 源代码的目录中执行如下命令：

```
→bitcoin git:(master) git checkout v0.12.1
→bitcoin git:(9779e1e) ✖
```

这样就准备好了 Bitcoin v0.12.1 的源代码。

4.2.1　主要模块

Bitcoin 客户端主要由如下几个模块组成。

（1）初始化和启动

在启动阶段，客户端执行多种初始化任务，最后启动多线程处理并发操作。

（2）P2P 网络

本地节点利用多种技术发现其他节点，在与之建立网络连接后，接收节点消息并利用 socket 发送消息到其他节点。

（3）区块交换

节点向其他节点广播自己存在的区块并互相交换区块，从而建立区块链 blockchain。节点在收到数据块的同时，会验证数据块是否合法，并将内存中与数据块重复的交易信息清除掉。

（4）交易交换

节点之间互相交换并传输交易，客户端把交易关联到本地钱包的比特币地址。交易信息会被广播到全网节点上，每个节点都会验证交易的前一个动作是否合法，如果合法，就将交易保存在内存中，等待进入数据区块 block。

（5）挖矿

挖矿（mining）指的是利用工作证明（Proof of Work）生产数据块的动作。

（6）钱包服务

- 客户端利用本地钱包创建交易。
- 客户端将交易与本地钱包的地址关联起来。
- 客户端提供管理本地钱包的服务。

（7）RPC 接口服务

客户端提供基于 HTTP 的 JSON-RPC 接口来执行多种操作功能并管理本地钱包。

（8）GUI

bitcoin-qt 提供图形用户操作界面，由于该部分不涉及比特币的核心代码逻辑，后续的代码分析将不会涉及这部分。

（9）数据目录

bitcoin 默认的数据目录见表 4-1。

表 4-1 默认的数据目录

操作系统	bitcoin 数据目录默认路径	bitcoin.conf 配置文件默认路径
Windows	%APPDATA%\Bitcoin\	C:\Users\username\AppData\Roaming\Bitcoin\bitcoin.conf
Linux	$HOME/.bitcoin/	/home/username/.bitcoin/bitcoin.conf
Mac OSX	$HOME/Library/Application Support/Bitcoin/	/Users/username/Library/Application Support/Bitcoin/bitcoin.conf

该目录及子目录中的文件说明见表 4-2。

表 4-2 主要配置及数据文件

文件或目录	详细描述
bitcoin.conf	bitcoin 配置文件，bitcoind 启动的时候会读取这个文件
debug.log	调试信息文件，各种日志写入均存储在该文件中
peers.dat	节点的信息
wallet.dat	钱包文件，保存你的私钥和相关交易记录，非常重要
blocks	区块链（blockchain）的数据存储目录
chainstate	区块链（blockchain）的状态存储目录
testnet3	测试链的数据目录，在 bitcoin.conf 中配置 testnet = 1 即可使用测试网络。测试链具有不同的起始块（genesis block）。详见 https://en.bitcoin.it/wiki/Running_Bitcoin

主要入口函数所在的文件见表 4-3。

表 4-3 主要入口函数

文　件	重要函数
bitcoind.cpp	Main、AppInit
bitcoin-cli.cpp	Main、AppInitRPC、CommandLineRPC、CallRPC
bitcoin-tx.cpp	Main、AppInitRawTx、CommandLineRawTx、MutateTx、OutputTx
init.cpp	AppInit2

表 4-3 中提到的函数 AppInit2 中会开启线程组和多个独立线程，注意，所有的网络线程都在同一个线程组，由于 bitcoind 是多线程程序，因此也就意味着有多个函数在并发执行，表 4-4 是主要的线程和线程所在的文件，可通过分析函数里的层层调用，逐渐了解模块的内

部实现。

表 4-4　主要的线程

线　程	说　明	所在文件
ThreadScriptCheck	脚本检查	main.cpp
AppInitServers	RPC、REST 服务入口	init.cpp
ThreadImport	区块导入	init.cpp
TorControlThread	Tor 连接线程	torcontrol.cpp
StartNode	节点启动入口	net.cpp
Discover	节点发现	net.cpp
ThreadDNSAddressSeed	DNS 地址种子解析	net.cpp
ThreadSocketHandler	接受外部连接，socket 消息的接收与发送	net.cpp
ThreadOpenAddedConnections	-addnode 向外连接	net.cpp
ThreadOpenConnections	主动向外连接	net.cpp
ThreadMessageHandler	消息处理	net.cpp
BitcoinMiner	挖矿入口	miner.cpp
ThreadFlushWalletDB	周期性刷新钱包数据到存储文件	walletdb.cpp

4.2.2　初始化和启动

在分析代码时，首先要找到 bitcoind 的代码入口，它在 bitcoind.cpp 中的 179 行，如下：

```
int main(int argc, char* argv[])
{
  SetupEnvironment();

  // Connect bitcoind signal handlers
  noui_connect();

  return (AppInit(argc, argv) ? 0 : 1);
}
```

在 main() 函数中，SetupEnvironment() 用来准备环境，noui_connect() 用来链接 bitcoind 的信号处理，AppInit() 进行程序的基本初始化，AppInit() 主要要做以下几件事。

- 调用 ParseParameters 解析命令行参数。
- 读取 bitcoind.conf 文件，解析配置。
- 若以 bitcoind-daemon 模式启动，则通过 fork() 创建后台进程。
- 调用 InitLogging 配置日志。
- 调用 InitParameterInteraction 配置参数。
- 然后进入 AppInit2 函数，进行第二步的核心初始化。

AppInit2 在 init.cpp 的 786 行，AppInit2 里包含了 Bitcoin 的大部分初始程序，包括读取区块索引、加载区块、加载钱包，以及初始化其他线程，该过程分 12 步进行，具体如下：

```
/** 初始化 bitcoin.
 *  @pre 解析参数并读取配置文件。
 */
bool AppInit2(boost::thread_group& threadGroup, CScheduler& scheduler)
{
  // ************* Step 1: setup(安装网络环境，挂接事件处理器等。)
  // ************* Step 2: parameter interactions(进一步的参数交互设置，如区块裁剪 prune
和 txindex 的冲突检查、文件描述符的限制检查等。)
  // ************* Step 3: parameter-to-internal-flags(参数转换为内部变量，这样外部参数
的设置将转换成程序内部的状态。)

  // ************* Step 4: application initialization: dir lock, daemonize, idfile,
debug log(初始化 ECC，目录锁检查，保证只有一个 bitcoind 运行等。)
  // ************* Step 5: verify wallet database integrity(若启用钱包功能，则会检查钱
包数据库的完整性。)
  // ************* Step 6: network initialization(网络初始化。)
  // ************* Step 7: load block chain(加载区块链数据，即 blocks 目录下的数据。)
  // ************* Step 8: load wallet(若启用钱包功能，则加载钱包。)
  // *************Step 9: data directory maintenance(若是裁剪模式，则进行 blockstore
的裁剪。)
  // ************* Step 10: import blocks(导入数据块。)
  // ************* Step 11: start node(启动节点服务，监听网络 p2p 请求，若启用 -gen 挖矿参数，
则调用 GenerateBitcoins 启动数个挖矿线程 BitcoinMiner。)
  // ************* Step 12: finished(完成。)
}
```

4.2.3 P2P 网络

1. 网络处理线程

P2P 网络部分主要包含在 net.cpp 和 netbase.cpp 文件中，netbase.cpp 内主要是一些辅助函数，因此这里主要分析 net.cpp 文件中的函数，具体如下。

- 函数 ThreadDNSAddressSeed 负责从 DNS seed 解析出 IP 地址，并加入 IP 地址管理器 CAddrMan 中。
- 函数 ThreadOpenConnections 负责连接其他比特币节点的 IP，包括通过 -connect 直接配置的 IP 地址，以及通过 DNS seed 解析出的 IP 地址。
- 函数 ThreadOpenAddedConnections 负责处理 -addnode 方式添加的节点连接。
- 函数 ThreadSocketHandler 负责监听端口来接受其他节点的进入连接，把无用节点进行断开处理，以及节点的消息处理。

2. 选择节点地址的规则

如果用户通过 -connect 配置了地址，则程序只会连接这些 IP 节点。在连接好这些节点后，程序进入 500ms 的死循环。连接节点的逻辑在 OpenNetworkConnection 函数中，若 FindNode 发现连接已经建立，则直接返回 OpenNetworkConnection，否则就会通过 ConnectNode 建立连接生成 CNode 对象，并把 CNode 加入数组 vector<CNode*>vNodes。

3. 连接数限制

程序的向外连接数默认最大值为 MAX_OUTBOUND_CONNECTIONS，默认 8 个，通过 CSemaphore 进行流控。

程序的接入连接数默认值最多为 120 个，计算方式如下：

```
DEFAULT_MAX_PEER_CONNECTIONS-MAX_OUTBOUND_CONNECTIONS = 128-8=120
```

4. 消息处理

- ThreadSocketHandler 遍历 vNodes 连接向量，处理 socket 中数据的接收与发送。ReceiveMsgBytes 接收数据解析出一条完整的消息后插入消息接收队列 vRecvMsg。
- SendMessages 把消息插入发送队列 vSendMsg，SocketSendData 遍历消息发送队列 vSendMsg 发送数据。
- ThreadMessageHandler 通过调用 ProcessMessages 处理接收到的消息（用信号建立联系）。
- ProcessMessages 是比特币协议的核心处理入口部分，它会解析消息头，然后分发到 ProcessMessage 中进行具体消息的处理。
- SendMessages 是消息的发送部分，内部调用 PushMessage 进行数据打包。


注意　ProcessMessages 和 SendMessages 不在 net.cpp 中，而是在 main.cpp 中。

5. 网络类型

表 4-5 给出了网络类型及相关信息。

表 4-5　网络类型

网　络	默认端口	魔　数	最大 nBits
Mainnet	8333	0xf9beb4d9	0x1d00ffff
Testnet	18333	0x0b110907	0x1d00ffff
Regtest	18444	0xfabfb5da	0x207fffff

6. 消息报文格式

表 4-6 给出了消息报文格式。

表 4-6　消息报文格式

字　段	大　小	说　明
message start	4	魔数 magic，鉴别网络类型
command	12	消息类型，不足长度全部补 \0，比如 version\0\0\0\0\0
payload size	4	数据长度，消息长度最大为 MAX_PROTOCOL_MESSAGE_LENGTH = 2MB
checksum	4	校验和，SHA256（SHA256（payload））的前 4 个字节，对于 VERACK、GETADDR 和 SENDHEADERS 这种无 payload 的消息，则固定为 0x5df6e0e2（SHA256（SHA256（<empty string>）））
payload	x	数据。注意：比特币的消息报文中，绝大多数整数都是使用的 little-endian 小端编码，只有 IP 地址或端口号使用 big-endian 大端编码。

7. 消息类型

图 4-1 给出了消息协议流的概览。其中消息类型定义在 allNetMessageTypes 数组变量里，具体说明见表 4-7。

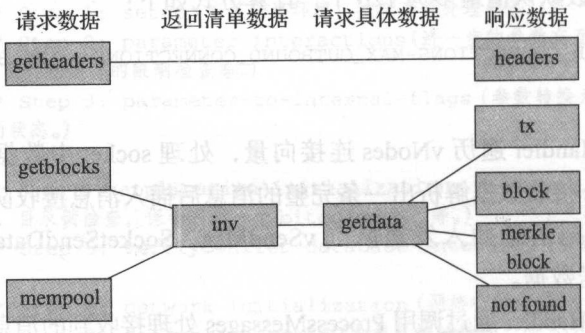


图 4-1 比特币 P2P 协议请求和响应消息概览

表 4-7 消息类型

消息类型	说 明
VERSION	当主动连接上对方时，发送 version 消息；监听方只有收到 version 消息时才会回复 version 和 verack 消息
VERACK	version ACK 版本确认消息
ADDR	转发网络上的节点地址列表消息
GETADDR	主动请求节点回复一个 addr 消息，以便快速更新本地地址库
GETBLOCKS	发送此消息以期返回一个包含编号从 hash_start 到 hash_stop 的 block 列表的 inv 消息。若 hash_start 到 hash_stop 的 block 数超过了 500，则在 500 处截止。欲获取后面的区块 Hash，需要重新发送 getblocks 消息
GETHEADERS	获取包含编号 hash_start 到 hash_stop 的至多 2000 个 block 的 header 包。要获取之后的区块 Hash，需要重新发送 getheaders 消息。这个消息用于快速下载不包含相关交易的 blockchain
INV	节点通过此消息宣告（advertise）它所拥有的对象信息。“我有这些 blocks/txs...”，一般当一个新块或交易转发时会主动发送这个消息，也可以用于应答 getblocks 消息
HEADERS	返回 block 的头部以应答 getheaders
GETDATA	getdata 用于应答 inv 消息来获取指定的对象，它通常在接收到 inv 包并滤去已有元素后发送到对方节点以获取未有元素。对方收到 getdata 消息后，回复 block 或 tx 消息
SENDHEADERS	指示节点优先用 headers 消息代替 inv 消息接收新块通告。新添加于 BIP130、Bitcoin Core 0.12.0、protocol version 70012
TX	回复 getdata 消息，发送 tx 内容
BLOCK	回复 getdata 消息，发送区块内容
MEMPOOL	收集内存池交易
PING	检查连接是否在线
PONG	确认 ping 消息
ALERT	alert 消息用于在节点间转发通知，使其传遍整个网络，比如版本升级
FILTERLOAD	用于 Bloom filter

（续）

消息类型	说　明
FILTERADD	用于 Bloom filter
FILTERCLEAR	用于 Bloom filter
REJECT	告知对方节点上一（几）个消息被拒绝
NOTFOUND	收到 getdata 消息时，返回告知对方没有发现 tx 或 block

表 4-7 包含了消息的大部分类型，下面就来分别看看各种消息类型的说明。

（1）VERSION 消息

消息体说明见表 4-8。

表 4-8　VERSION 消息结构

字段尺寸	描　述	数据类型	说　明
4	version	uint32_t	节点使用的协议版本标识号
8	services	uint64_t	提供的服务（bitfield） NODE_NETWORK NODE_GETUTXO NODE_BLOOM 具体看 protocol.h 的 nServices flags
8	timestamp	uint64_t	以秒计算的标准 UNIX 时间戳
26	addr_me	net_addr	发送者地址信息
version >= 106			
26	addr_you	net_addr	接收者地址信息
8	nonce	uint64_t	节点的随机唯一 ID
?	sub_version_num	var_str	子版本字串
version >= 209			
4	start_height	uint32_t	发送节点拥有的最新 block
1	relay_txes/blocks_only	uint8_t	是否转发 tx invs

主动连接上对方节点时，发的第一条消息是 VERSION 消息，在 CNode 构造函数里代码片段如下：

```
// Be shy and don't send version until we hear
if (hSocket != INVALID_SOCKET && !fInbound)
    PushVersion();
```

待节点收到 VERSION 消息后，可能会进行如下几种的操作。

- 若对方已经发送过一次，则回复一个 REJECT 消息，消息体为 REJECT_DUPLICATE。
- 若对方的协议 version<MIN_PEER_PROTO_VERSION，则回复一个 REJECT 消息，消息体为 REJECT_OBSOLETE。然后连续回复 VERSION 消息、VERACK 消息。
- 若自己是监听节点，则回复 ADDR 消息发送自己的地址和端口；若对方为 fOneShot，大于 CADDR_TIME_VERSION 或本地地址库不足 1000 个，则回复 GETADDR 消息。

❑ 若本地接收过告警消息，则转发 ALERT 消息。

最后，记录下对方时间戳和本地时间戳之差，到此，网络的通道层正式建立连接。

若节点不发送 VERSION 消息就发送其他消息，但只要超过 DEFAULT_BANSCORE_THRESHOLD = 100 次会被禁止掉。

（2）VERACK 消息

当收到 VERACK 消息时，设置对方节点状态为“连接成功”，若对方节点协议版本大于 SENDHEADERS_VERSION = 70012，则发送 SENDHEADERS 消息。

（3）ADDR 消息

ADDR 消息用于转发网络上的节点地址列表。

其消息体 payload 的结构如表 4-9 所示。

表 4-9 ADDR 消息结构

字段尺寸	描 述	数据类型	说 明
1+	count	var_int	IP 地址数，最大 1000
30x?	addr_list	addr_vect[]	IP 地址入口

其中，IP 地址入口字段的结构如表 4-10 所示。

表 4-10 IP 地址入口结构

字段尺寸	描 述	数据类型	说 明
4	time	uint32	节点公告 IP 地址的时间戳
8	services	uint64_t	同 version 节点的 services
16	IP address	char	IP 地址，大端子节序
2	port	uint16_t	IP 端口，大端子节序

（4）GETADDR 消息

GETADDR 消息会主动请求节点回复一个 addr 消息，以便快速更新本地地址库。它没有消息体 payload。

（5）GETBLOCKS 消息

发送此消息以期返回一个包含编号从 hash_start 到 hash_stop 的区块列表的 inv 消息。若 hash_start 到 hash_stop 的区块数超过 500，则在 500 处截止。欲获取后面的区块 Hash，则需要重新发送 getblocks 消息。

其消息体 payload 的结构如表 4-11 所示。

表 4-11 GETBLOCKS 消息结构

字段尺寸	描 述	数据类型	说 明
4	version	uint32_t	同 version 节点的协议版本号 version
1+	count	var_int	hash_start 的数量，一般为 1 ~ 200

（续）

字段尺寸	描　述	数据类型	说　明
32+	hash_start	char[32]	发送节点已知的最新区块 Hash
32	hash_stop	char[32]	请求的最后一个区块的 Hash，若要获得尽可能多的区块则设为 0，最多返回 500 个

（6）GETHEADERS 消息

GETHEADERS 消息用于获取包含编号 hash_start 到 hash_stop 且至多 2000 个区块的 header 包。要获取之后的区块 Hash，需要重新发送 getheaders 消息。这个消息用于快速下载不包含相关交易的区块链（blockchain）。

其消息体 payload 的结构及说明如表 4-12 所示。

表 4-12　GETHEADERS 消息结构

字段尺寸	描　述	数据类型	说　明
4	version	uint32_t	同 version 节点的协议版本号 version
1+	hash 数	var_int	hash_start 的数量
32+	hash_start	char[32]	发送节点已知的最新区块 Hash
32	hash_stop	char[32]	请求的最后一个区块的 Hash，若要获得尽可能多的区块则设为 0，最多返回 2000 个

（7）INV 消息

INV 消息用于发送本节点的交易和区块列表。

其消息体 payload 的结构及说明如表 4-13 所示。

表 4-13　INV 消息结构

字段尺寸	描　述	数据类型	说　明
1+	Count	var_int	清单（inventory）数量
36x?	inventory	inv_vect[]	清单（inventory）数据

其中，inventory 字段的结构见表 4-14。

表 4-14　inventory 结构

字段尺寸	描　述	数据类型	说　明
4	type	int	清单（inventory）类型：MSG_TX MSG_BLOCK MSG_FILTERED_BLOCK
32	hash	uint256 / char[32]	清单（inventory）hash 值，SHA256（SHA256（object））

（8）HEADERS

headers 消息用于返回区块的头部以应答 getheaders。

其消息体 payload 的结构如表 4-15 所示。

表 4-15 HEADERS 消息结构

字段尺寸	描 述	数据类型	说 明
?	count	var_int	区块头数量，最大 2 000。当区块数为 2 000 时，继续发送 getheaders 消息以请求更多的 headers；当该数小于 2 000 时，则可以认为已经遍历到对方节点的区块链最新点了
81x?	headers	block_header_ex[]	区块头和 tx count

其中 block_header_ex 定义见表 4-16。

表 4-16 block_header_ex 结构

字段尺寸	描 述	数据类型	说 明
4	version	uint32_t	block 版本信息，基于创建该区块的软件版本
32	prev_block	char[32]	该区块前一区块的 Hash
32	merkle_root	char[32]	与该区块相关的全部交易之 Hash（Merkle 树），Merkle 树是 Hash 的二叉树，在比特币中使用两次 SHA-256 算法来生成 Merkle 树，如果叶子个数为奇数，则要重复计算最后一个叶子的两次 SHA-256 值，以达到偶数叶子节点的要求
4	timestamp	uint32_t	记录 block 创建时间的时间戳
4	bits	uint32_t	创建 block 的计算难度
4	nonce	uint32_t	用于生成这一 block 的 Nonce 值
1	txn_count	uint8_t	交易数，这个值总是 0

（9）GETDATA 消息

该消息的内容同 INV 消息，待收到对方节点的 INV 消息后，返回 GETDATA 消息到对方节点去获取节点自己不存在的 block 或 tx。

（10）SENDHEADERS 消息

SENDHEADERS 消息指示节点优先用 headers 消息代替 inv 消息接收新块通知，它没有消息体 payload。

（11）BLOCK 消息

block 消息用于响应请求交易信息的 getdata 消息。

其消息体 payload（注意和 HEADERS 消息体的区别：txn_count、txns）的结构及说明如表 4-17 所示。

表 4-17 BLOCK 消息结构

字段尺寸	描 述	数据类型	说 明
4	version	uint32_t	block 版本信息，基于创建该 block 的软件版本
32	prev_block	char[32]	该 block 前一 block 的 Hash
32	merkle_root	char[32]	与该 block 相关的全部交易之 Hash（Merkle 树）
4	timestamp	uint32_t	记录 block 创建时间的时间戳
4	bits	uint32_t	这一 block 的计算难度
4	nonce	uint32_t	用于生成这一 block 的 nonce 值
?	txn_count	var_int	交易数量
?	txns	tx[]	交易，以 tx 格式存储

一个 block 的第一笔交易必须是生成比特币的交易，它不包含任何输入交易，而是生成比特币，这些比特币通常被完成这个 block 的人获得。这样的交易被称作“ coinbase 交易”。由于每个 block 只有一个 coinbase 交易，因此无须执行脚本即可被 bitcoin 客户端接受。

如果一笔交易不是 coinbase 交易，那么它会引用前一笔交易的 Hash 和其他交易的输出作为这笔交易的输入，并执行这笔交易输入部分的脚本。然后引用交易的输出部分的脚本也会被执行。如果栈顶的元素为真则交易被认可。

（12）TX 消息

TX 消息针对一笔比特币交易进行描述，用于应答 getdata 消息。

其消息体 payload 的结构及说明如表 4-18 所示。

表 4-18　TX 消息结构

字段尺寸	描　述	数据类型	说　明
4	version	uint32_t	交易数据格式版本
1+	tx_in count	var_int	交易的输入数
41+	tx_in	tx_in[]	交易输入或比特币来源列表
1+	tx_out count	var_int	交易的输出数
8+	tx_out	tx_out[]	交易输出或比特币去向列表
4	lock_time	uint32_t	

其中，lock_time 为锁定交易的期限或 block 数目。若该交易的所有输入 tx_in 的 sequence 字段为 uint32_t 最大值（0xFFFFFFFF），则忽略该字段的逻辑检查。

- 当 sequence < 0xFFFFFFFF，且 lock_time == 0 时，该交易可立即被打包。
- 当 sequence < 0xFFFFFFFF，且 lock_time != 0 时：
  - 若 lock_time < 500000000，则 lock_time 代表区块数，该交易只能被打包进高度大于等于 lock_time 的区块。
  - 若 lock_time >= 500000000，则 lock_time 代表 Unix 时间戳，该交易只能等到当前时间大于等于 lock_time 时才能被打包进区块。

再来看看 tx_in 的构成（如表 4-19 所示），一个支付的输入用“旧”的 txid 和 output 索引（vout，即 output vector）来鉴别“钱”的来源。

表 4-19　tx_in 的结构

字段尺寸	描　述	数据类型	说　明
36	previous_tx_output	OutPoint	对前一输出的引用，即需要出示那个账单的 txid，也就是说，你花的任何一笔钱都应该有人转给你过
1+	script length	var_int	signature script 的长度
?	script signature	uchar[]	用于确认交易授权的计算脚本，你对这笔交易的签名，就是用你的私钥对 previous_output 的引用做散列，因为只有你能做这个散列，这宣誓了你的拥有权
4	sequence	uint32_t	发送者定义的交易版本，用于在交易被写入 block 之前更改交易

OutPoint 结构的构成如表 4-20 所示。

表 4-20 OutPoint 结构

字段尺寸	描 述	数据类型	说 明
32	hash	char[32]	交易 Hash
4	index	uint32_t	指定 tx 输出的索引，第一笔输出的索引是 0，以此类推

接收方 tx_out 的构成见表 4-21。

表 4-21 tx_out 结构

字段尺寸	描 述	数据类型	说 明
8	value	uint64_t	要发多少，交易的比特币数量（单位是 0.00000001）
1+	pk_script length	var_int	pk_script 的长度
?	pk_script	uchar[]	一般为对方的公钥，比特币账户就是一段公钥，script 由一系列与交易相关的信息和操作组成，详情请参考 script.h 和 script.cpp，分析在脚本系统一节

（13）MEMPOOL 消息

MEMPOOL 消息用于收集内存池交易，它没有消息体 payload。

（14）PING 消息

PING 消息用于检查连接是否在线。

其消息体 payload 的结构及说明如表 4-22 所示。

表 4-22 PING 消息结构

字段尺寸	描 述	数据类型	说 明
8	nonce	uint64_t	随机数

（15）PONG 消息

消息内容同 PING 消息，待收到对方节点的 PING 消息后回复 PONG 消息，表明连接还在线。

（16）ALERT 消息

alert 消息用于在节点间发送通知使其传遍整个网络。如果签名验证这个 alert 来自 Bitcoin 的核心开发组，建议将这条消息显示给终端用户。交易尝试，尤其是客户端间的自动交易则建议停止。消息文字应当记入记录文件并传到每个用户。

其消息体 payload 的结构及说明如表 4-23 所示。

表 4-23 ALERT 消息结构

字段尺寸	描 述	数据类型	说 明
?	Message	var_str	向网络中所有节点发出的系统消息
?	Signature	var_str	可由公钥验证 Satoshi 授权或创建了此信息的签名

4.2.4　交易和区块

4.2.3 节已经分析了区块和交易在协议中的结构，可以看到，block 和 tx 的定义在 primitives 文件夹中。

交易就是将你的比特币地址余额支付给某个比特币地址。简单地说就是钱从哪里来就花到哪里去，类似会计的三重记账法。

待签名一个账单并把签单发送到全世界以后，所有收到这个单子的客户端都会效验你这个单子对不对，比如会效验你的签名，是不是你发的，以及你是否有那么多钱等。

如果大家算过这个交易没问题，那么基本上就算转账成功了，等待打包进入区块，进而进入区块链。区块即是收集上面广播的 tx 并带上挖矿的 coinbase tx，它们会作为一个整体纪录等待被“挖矿”记入区块链。

具体到每笔交易，又分为以下几种类型。

- 支付到比特币一般地址的交易 Pay-to-PubkeyHash Tx .(P2PKH Tx)。
- 支付到比特币多签地址的交易 Pay-to-Script-Hash Tx .(P2SH Tx)。
- 比特币挖矿生成的交易 Generation coinbaseTx .(Coinbase Tx)。

以上各种交易的验证依赖于脚本系统，4.2.5 节会进一步说明。

4.2.5　脚本系统

脚本的本质就是实现逻辑的动态控制过程，比如游戏中经常用 lua 脚本控制游戏逻辑。比特币使用的脚本系统是一个基于栈的，从左到右的，为了安全考虑特意设计成没有循环的非图灵完备系统。

比特币的脚本系统主要用于对交易的当前输入及其上一个输出进行身份校验，从而确认该笔交易是否有效。上一笔的输出脚本就是拥有者宣告谁能消费这笔钱的锁，当前交易的输入脚本就是打开这把锁的钥匙，从而宣告自己拥有这笔钱，进而完成这笔钱的转移，即交易的核心。脚本系统就是智能合约，使用脚本系统的比特币则称为第一种可编程的货币。

脚本的执行需要解析引擎，主要代码在 script 文件夹中，下面以最为常用的典型的脚本 P2PKH 为例，从概念上分析整个脚本的执行过程。

先来看看 P2PKH 脚本的组成元素。

- 公钥脚本：OP_DUP OP_HASH160 <PubkeyHash> OP_EQUALVERIFY OP_CHECKSIG
- 签名脚本：<sig><pubkey>

在送入脚本解析器之前，要把签名脚本附在公钥脚本的前面，整体构成代码如下：

```
<Sig><PubKey> OP_DUP OP_HASH160 <PubkeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

这里的 OP_ 是 operation 操作数的缩写，以 OP_ 开头的都是操作数。

表 4-24 展示的是脚本状态的迁移过程。

表 4-24 脚本状态迁移过程

步　骤	操作动作	脚本状态迁移	栈状态迁移
初始状态	初始无操作	初始脚本： <Sig><PubKey> OP_DUP OP_HASH160 <PubkeyHash> OP_EQUALVERIFY OP_CHECKSIG	初始栈为空 []
第 1 步	把 Sig 签名数据压入栈顶，即压栈 <Sig>	剩余脚本： <PubKey> OP_DUP OP_HASH160 <PubkeyHash> OP_EQUALVERIFY OP_CHECKSIG	栈变为： [<Sig>]
第 2 步	把 PubKey 数据压入栈顶，即压栈 <PubKey>	剩余脚本： OP_DUP OP_HASH160 <PubkeyHash> OP_EQUALVERIFY OP_CHECKSIG	栈变为： [<PubKey>] [<Sig>]
第 3 步	执行 OP_DUP 复制栈顶元素	剩余脚本： OP_HASH160 <PubkeyHash> OP_EQUALVERIFY OP_CHECKSIG	栈变为： [<PubKey>] [<PubKey>] [<Sig>]
第 4 步	把栈顶元素也就是 PubKey 做 OP_HASH160 元计算后得到 <PubKeyHashX>	剩余脚本： <PubkeyHash> OP_EQUALVERIFY OP_CHECKSIG	栈变为： [<PubKeyHashX>][<PubKey>] [<Sig>]
第 5 步	把剩余脚本里的第一个元素数据压入栈顶，即压栈 <PubkeyHash>	剩余脚本： OP_EQUALVERIFY OP_CHECKSIG	栈变为： [<PubKeyHash>] [<PubKeyHashX>][<PubKey>] [<Sig>]
第 6 步	OP_EQUALVERIFY 检查栈顶前面两个元素是否相等，如果相等则继续执行脚本，否则中断执行	剩余脚本： OP_CHECKSIG	栈变为： [<PubKey>] [<Sig>]
第 7 步	OP_CHECKSIG 校验签名，返回 Result。至此，整个脚本执行完毕	剩余脚本：空	栈变为：[<Result>]

4.2.6　挖矿

挖矿就是用 Hashcash 概念打包区块 block 至区块链 blockchain，并自生新币 coinbase 的过程。挖矿有两个目的：一是产生新币，二是打包转移旧币。挖矿逻辑中有一个核心概念就是 Hashcash，理解了它就理解了挖矿。

Hashcash 的原理如图 4-2 所示。

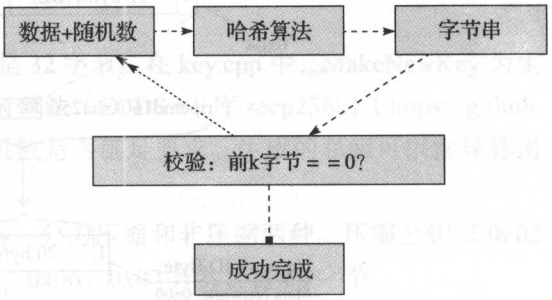


图 4-2　Hashcash 原理

生产方：一串含有信息的字节串，首先产生一个随机数 R，使用某种运算将之合并，再进行散列（如 SHA-256）；若散列后的比特串的头 K 位全部是 0，那么过程结束，发送原始的字节串及找到的随机数 R 至验证方；否则，重新生成随机数 R。

验证方：简单地验证收到的比特串和随机数 R，检验经过散列后的比特串的头 K 位是否全部是 0；若是则有效。

通过数论及密码学设计，可以保证至少存在 1 个 $P \in [1, 2K]$ 使得其满足头 K 位全部是 0。而要寻找这一神奇的 R，只能通过暴力破解来实现。

不难看出，Hashcash 具有如下性质。

- 理论上，其计算复杂度为指数级，需要进行大量的 Hash 运算，而实际上执行 Hashcash 过程所需要的计算量，和 R 的大小直接相关，所以 Hashcash 的计算难度是可调节的。
- 验证阶段的计算量和 R 的大小无关。

Bitcoin 使用的 PoW 机制与 Hashcash 的方式类似。但是也存在如下几点变化。

- Hashcash 通常只能将难度（difficulty）翻倍或减半，而 Bitcoin 则有更为复杂的调整策略。
- Hashcash 通常使用 1 轮 SHA-1 算法，而 Bitcoin 的每个开采都需要 2 轮 SHA-256 计算。
- 用于识别每个 block 的 SHA-256 使用 block 消息结构的前 6 个字段计算（version、prev_block、merkle_root、timestamp、bits、nonce，后接标准 SHA-256 填充，共 2 个 64 字节块），而非整个 block。计算 Hash 时 SHA-256 算法只需要处理 2 个块。由于 nonce 字段在第二个块里，在挖矿过程中，第一个块保持不变，因此只需要处理第二个块即可。

由于 Hash 运算需要消耗大量 CPU 时间，因此系统会定时发放 Bitcoin 以奖励贡献计算资源的系统维护者，Bitcoin 就此在网络中产生，这也是 Bitcoin 产生的唯一方式，称为挖矿（Mining）。

具体的代码请参考 4.3.1 节中的 ScanHash 函数，也可以在 miner.h、miner.cpp 文件中查看。

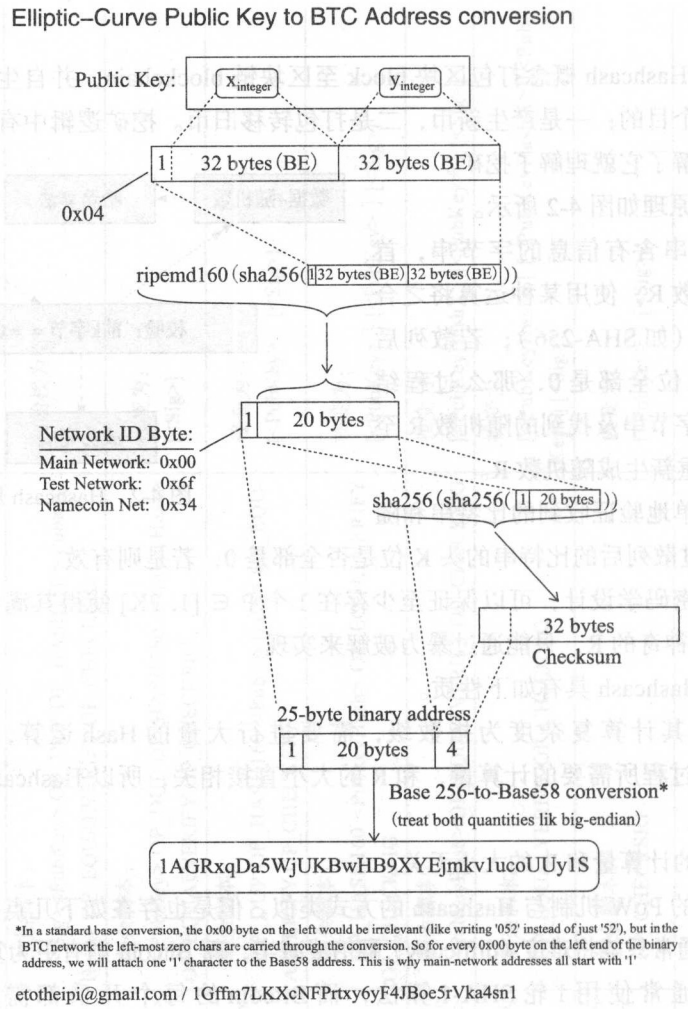


图 4-3 地址生成过程

4.2.7 私钥

开发者可以淡化钱包概念本身，因为钱包概念在比特币中不过是用户自身的私钥、地址、交易的存储地址。因为根据私钥可以导出公钥，公钥又可以导出地址，地址又可以在区块链中查询到交易，所以钱包的核心就是私钥的保存与使用。私钥就是你的钥匙，脚本系统保证你用这把钥匙去打开区块链里的一把或多把锁，从而宣告拥有的比特币余额。所以一定要保管好你的私钥，而不是你的钱包。

既然只需要保密好私钥就行了，那为什么还有很多比特币钱包提供商呢？简单地说他们就是提供更加安全方便的方案，比如 HD 分层钱包方案，就实现了方便的私钥记忆与权限分级功能。所以钱包功能的核心就是提供私钥的安全存储管理体系，其他的比如用 SPV 或

Fullnode 模式、移动端、Web 端、多签等概念都是基于此的延伸。所以，从纯技术的角度来说，钱包完全可以从比特币核心中剥离出来，因此本章也不多讲钱包体系，只专注于私钥→公钥→地址的推导。

在 key.h 文件里有私钥 CPrivKey 的定义，在封装私钥定义 CKey 中有内存存储的定义 unsigned char vch[32];

vch 用来存储 key 的数据，即私钥的长度是 32 字节。在 key.cpp 中，MakeNewKey 为生成新 key 的函数，内部调用椭圆曲线数字签名算法（ECDSA）库 secp256k1（https://github.com/bitcoin-core/secp256k1）来检查生成的随机数是否满足要求。生成的私钥可以推导算出（调用 GetPubKey）对应的唯一公钥。

在 pubkey.h 文件里有公钥的定义 CPubKey，分为压缩和非压缩两种。压缩公钥以 0x02 或 0x03 开头，占 33 字节；非压缩公钥以 0x04、0x06、0x07 开头，占 65 字节。

非压缩公钥可以按照图 4-3 的规则推导出比特币地址。对于压缩公钥，只需要取图 4-3 中 Public key 的 X 部分即可。

4.3　性能实战

4.3.1　建立私链

最简单的建立私有链（或创建山寨币）的方式就是改参数，一般在 chainparams.cpp 中有不同链的参数定义，如 main、test、regtest 等。

（1）修改创世块参数

首先是修改创世块 coinbase 的信息。

下面这段代码 CreateGenesisBlock 是创世块交易，中本聪用 2009 年 1 月 3 日报纸上的一个标题 pszTimestamp 作为 coinbase 的内容，为的是证明这个创世块的产生迟于 2009 年 1 月 3 日，以表明在这之前没有预挖比特币，以此作为时间凭证。

```
    static CBlock CreateGenesisBlock(uint32_t nTime, uint32_t nNonce, uint32_t
nBits, int32_t nVersion, const CAmount& genesisReward)
    {
      const char* pszTimestamp = "The Times 03/Jan/2009 Chancellor on brink of second
bailout for banks";
    const CScript genesisOutputScript = CScript() << ParseHex("04678afdb0fe554827
1967f1a67130b7105cd6a828e03909a67962e0ea1f61deb649f6bc3f4cef38c4f35504e51ec112de5c3
84df7ba0b8d578a4c702b6bf11d5f") << OP_CHECKSIG;
    return CreateGenesisBlock(pszTimestamp, genesisOutputScript, nTime, nNonce, nBits,
nVersion, genesisReward);
    }
```

建立创世块的代码如下：

```
    genesis = CreateGenesisBlock(1231006505, 2083236893, 0x1d00ffff, 1, 50 * COIN);
```

其中，1231006505 代表区块链发布的 Unix 时间，可以在这里把北京时间转换成为 Unix 时间。

2083236893 为 Nonce 值，Nonce 这个值的发现只能从 0 开始遍历，使得 PoW 挖矿的 Hash 满足最低难度 consensus.powLimit，Nonce 不会只有一个，但是只要找到一个能够满足 nBits 条件，即可获得奖励了。

接着是修改挖矿的难度，代码如下：

```
consensus.powLimit = uint256S("00000000ffffffffffffffffffffffffffffffffffffffffffffffffffffffff");
```

其中参数 0x1d00ffff 为 nBits，nBits 参数可以和 powHash 互相转换。consensus.powLimit 代表最低难度。

第三步是修改货币总量，代码如下：

```
consensus.nSubsidyHalvingInterval = 210000;
```

第四步是修改区块奖励的初始数额，代码如下：

```
genesis = CreateGenesisBlock(1231006505, 2083236893, 0x1d00ffff, 1, 50 * COIN);
```

这里的 50 × COIN 即初始区块奖励的数额。

第五步是修改难度调节周期，代码如下：

```
consensus.nPowTargetTimespan = 14 * 24 * 60 * 60; //两周调整一次难度，
consensus.nPowTargetSpacing = 10 * 60;  //区块时间，平均每 10 分钟出一个块，若时间太短，则会因为传播需要时间，区块链容易发生分叉。
consensus.nMinerConfirmationWindow = 2016; //nPowTargetTimespan / nPowTargetSpacing
```

（2）修改网络协议头的魔数

pchMessageStart 是协议头里的魔数值，可以修改为任意的四个字节，代码如下：

```
pchMessageStart[0] = 0xf9;
pchMessageStart[1] = 0xbe;
pchMessageStart[2] = 0xb4;
pchMessageStart[3] = 0xd9;
```

（3）修改网络监听端口

```
nDefaultPort = 8333;//chainparams.cpp 中
nRPCPort = 8332;//chainparamsbase.cpp 中
```

（4）修改种子连接定义

```
vSeeds.push_back(CDNSSeedData("bitcoin.sipa.be", "seed.bitcoin.sipa.be")); // Pieter Wuille
vSeeds.push_back(CDNSSeedData("bluematt.me", "dnsseed.bluematt.me")); // Matt Corallo
vSeeds.push_back(CDNSSeedData("dashjr.org", "dnsseed.bitcoin.dashjr.org")); // Luke Dashjr
```

```
    vSeeds.push_back(CDNSSeedData("bitcoinstats.com", "seed.bitcoinstats.com")); //
Christian Decker
      vSeeds.push_back(CDNSSeedData("xf2.org", "bitseed.xf2.org")); //Jeff Garzik
    vSeeds.push_back(CDNSSeedData("bitcoin.jonasschnelli.ch", "seed.bitcoin.
jonasschnelli.ch")); //Jonas Schnelli
```

（5）修改币地址的前缀

```
      base58Prefixes[PUBKEY_ADDRESS] = std::vector<unsigned char>(1,0);
        base58Prefixes[SCRIPT_ADDRESS] = std::vector<unsigned char>(1,5);
        base58Prefixes[SECRET_KEY] =     std::vector<unsigned char>(1,128);
    base58Prefixes[EXT_PUBLIC_KEY] = boost::assign::list_of(0x04)(0x88)(0xB2)
(0x1E).convert_to_container<std::vector<unsigned char>>();
        base58Prefixes[EXT_SECRET_KEY] = boost::assign::list_of(0x04)(0x88)(0xAD)(0xE4).
convert_to_container<std::vector<unsigned char>>();
```

（6）coinbase 成熟确认数

该常数在 consensus.h 中定义，代码如下：

```
    /** Coinbase transaction outputs can only be spent after this number of new
blocks (network rule) */
    static const int COINBASE_MATURITY = 100;
```

（7）修改 checkpoint

```
    checkpointData = (CCheckpointData) {        };
```

这里可以填入创始块的 check points，其主要目的是防止网络分叉，现在我们还没有值，启动一次 bitcoind 后，即可从日志里找到这个值并填入。

（8）修改 publickey

```
    vAlertPubKey = ParseHex("04302390343f91cc401d56d68b123028bf52e5fca1939df127f63c
6467cdf9c8e2c14b61104cf817d0b780da337893ecc4aaff1309e536162dabbdb45200ca2b0a");
```

（9）修改 scriptPubKey

```
    const CScript genesisOutputScript = CScript() << ParseHex("04678afdb0fe55482719
67f1a67130b7105cd6a828e03909a67962e0ea1f61deb649f6bc3f4cef38c4f35504e51ec112de5c384
df7ba0b8d578a4c702b6bf11d5f") << OP_CHECKSIG;
```

（10）修改挖矿算法

bitcoind 自带的挖矿算法定义在 miner.cpp 文件的 ScanHash 中，可以根据需要进行修改。

ScanHash 函数的原理即保持 block 消息结构的前 5 个字段（version、prev_block、merkle_root、timestamp、bits）76 字节不变，通过遍历随机数 Nonce，然后把两者的数据拼接后做 Hash 运算，若 Hash 运算后的比特串的头 K 个字节全部都是 0，那么过程结束，返回挖矿成功，否则返回失败。

```
    bool static ScanHash(const CBlockHeader *pblock, uint32_t& nNonce, uint256 *phash)
    {
```

```
    //Write the first 76 bytes of the block header to a double-SHA256 state.
    CHash256 hasher;
    CDataStream ss(SER_NETWORK, PROTOCOL_VERSION);
    ss << *pblock;
    assert(ss.size() == 80);
    hasher.Write((unsigned char*)&ss[0], 76);

    while (true) {
      nNonce++;

      //Write the last 4 bytes of the block header (the nonce) to a copy of
      //the double-SHA256 state, and compute the result.
      CHash256(hasher).Write((unsigned char*)&nNonce, 4).Finalize((unsigned char*)
      phash);

      //Return the nonce if the hash has at least some zero bits,
      //caller will check if it has enough to reach the target
      if (((uint16_t*)phash)[15] == 0)
        return true;

      //If nothing found after trying for a while, return -1
      if ((nNonce & 0xfff) == 0)
        return false;
    }
  }
```

4.3.2 优化改进

Bitcoin Core（bitcoind）的原始设计效率有问题，并不适合做 Bitcoin Server，这里简单总结一下改进点，由于每一项改进都可以抽出来具体描述，限于篇幅具体内容请看参考以下信息。

- 剥离钱包：参考 https://github.com/btcsuite/btcd 的设计。
- 连接改进：提高连接节点总数量。
- 存储查询：blockchain 数据库替换优化，参考基于 PostgreSQL 的区块浏览器开源实现。https://github.com/haobtc/openblockchain。
- 协议优化：压缩区块（Compact Block），隔离认证 Segregated Witness，已被添加到 Bitcoin v0.13 版本。
- 传输优化：快速传播 block，参考 https://github.com/bitcoinfibre/bitcoinfibre。
- 链外优化：闪电网络，参考 http://lightning.network/ 和 https://github.com/ElementsProject/lightning。
- 挖矿优化：Stratum 协议，参考 https://en.bitcoin.it/wiki/Stratum_mining_protocol。
- 匿名优化：ZCASH 协议，参考 https://z.cash/。

4.4 API 开发

4.4.1 命令行调用

可以通过命令行或配置文件传入参数启动 bitcoind，如表 4-25 所示。

表 4-25　命令行选项

比特币核心版本 v0.12.1 使用：bitcoin-qt 或 bitcoind [命令行选项]	
选项：	
-?	本帮助信息，它会提示常用的命令行参数并退出
-version	打印版本然后退出
-alerts	收到并且显示 P2P 网络的告警（默认为 0）
-alertnotify = <cmd>	当收到相关提醒，或者看到一个长分叉时执行命令（%s 将替换为消息）
-blocknotify = <cmd>	当最佳数据块变化时执行命令（命令行中的 %s 会被替换成数据块 Hash 值）
-checkblocks = <n>	启动时检测多少个数据块（默认值为 288，0 = 所有）
-checklevel = <n>	数据块验证严密级别 -checkblocks（取值范围为 0-4，默认为 3）
-conf = <file>	指定配置文件（默认为 bitcoin.conf）
-datadir = <dir>	指定数据目录
-dbcache = <n>	设置以 MB 为单位的数据库缓存大小（4 到 16384，默认值为 100）
-loadblock = <file>	启动时从 blk000??.dat 文件导入数据块
-maxorphantx = <n>	内存中最多保留 <n> 笔孤立的交易（默认为 100）
-maxmempool = <n>	保持交易内存池大小低于 <n>MB（默认为 300）
-mempoolexpiry = <n>	内存池中保留交易不长于 <n> 小时（默认为 72）
-par = <n>	设置脚本验证的程序（取值范围为 –2 到 16，0 = 自动，< 0 = 保留自由的核心，默认值为 0）
-pid = <file>	指定 pid 文件（默认为 bitcoind.pid）
-prune = <n>	通过修剪（删除）旧数据块减少存储需求。此模式将禁用钱包支持，并与 -txindex 和 -rescan 不兼容。警告：还原此设置需要重新下载整个数据链。（默认值为 0 = 禁用修剪数据块，> 550 = 数据块文件目标大小，单位为 MiB）
-reindex	启动时重新为当前的 blk000??.dat 文件建立索引
-sysperms	创建系统默认权限的文件，而不是 umask 077（只在关闭钱包功能时有效）
-txindex	维护一份完整的交易索引，用于 getrawtransaction RPC 调用（默认值为 0）
连接选项：	
-addnode = <ip>	添加节点并与其保持连接
-banscore = <n>	与行为异常的节点断开连接的临界值（默认值为 100）
-bantime = <n>	重新允许行为异常的节点连接所间隔的秒数（默认值为 86400）
-bind = <addr>	绑定指定的 IP 地址开始监听。IPv6 地址请使用 [host]:port 格式
-connect = <ip>	仅连接到指定节点
-discover	发现自己的 IP 地址（默认：监听并且无 -externalip 或 -proxy 时为 1）
-dns	使用 -addnode、-seednode 和 -connect 选项时允许查询 DNS（默认为 1）
-dnsseed	使用 DNS 查找节点（默认值为 1）

（续）

比特币核心版本 v0.12.1 使用：bitcoin-qt 或 bitcoind [命令行选项]	
-externalip = <ip>	指定您的公共地址
-forcednsseed	始终通过 DNS 查询节点地址（默认为 0）
-listen	接受来自外部的连接（默认：如果不带 -proxy or–connect 则参数设置为 1）
-listenonion	自动创建 Tor 洋葱隐藏服务（默认为 1）
-maxconnections = <n>	保留最多 <n> 条节点连接（默认为 125）
-maxreceivebuffer = <n>	每个连接的最大接收缓存，<n> × 1000 字节（默认为 5000）
-maxsendbuffer = <n>	每个连接的最大发送缓存，<n> × 1000 字节（默认为 1000）
-onion = <ip:port>	通过 Tor 隐藏服务连接节点时使用不同的 SOCKS5 代理（默认为 -proxy）
-onlynet = <net>	只连接 <net> 网络中的节点（ipv4、ipv6 或 onion）
-permitbaremultisig	是否转发非 P2SH 格式的多签名交易（默认为 1）
-peerbloomfilters	支持利用布隆过滤器过滤区块和交易（默认为 1）
-port = <port>	使用端口 <port> 监听连接（默认为 8333；testnet: 18333）
-proxy = <ip:port>	通过 SOCKS5 代理连接
-proxyrandomize	为每个代理连接随机化凭据。这将启用 Tor 流隔离（默认为 1）
-seednode = <ip>	连接一个节点并获取对端地址，然后断开连接
-timeout = <n>	指定连接超时毫秒数（最小为 1，默认为 5000）
-torcontrol = <ip>:<port>	洋葱控制端口（默认为 127.0.0.1:9051）
-torpassword = <pass>	洋葱控制端口密码（默认为空）
-upnp	使用 UPnp 映射监听端口（默认为 0）
-whitebind = <addr>	绑定到指定地址和连接的白名单节点。IPv6 使用 [主机]：端口格式
-whitelist = <netmask>	节点白名单，网络掩码或 IP 址。可多次指定。白名单节点不能被 DoS 禁止，且转发所有来自于他们的交易（即便这些交易已经存在于 mempool 中），常用于网关
-whitelistrelay	非转发交易模式下也接受转发从白名单节点收到的交易（默认为 1）
-whitelistforcerelay	强制转发从白名单节点收到的交易，即使违反本地转发策略（默认为 1）
-maxuploadtarget = <n>	尝试保持上传带宽低于（MiB/24h），0 = 无限制（默认为 0）
钱包选项：	
-disablewallet	不要加载钱包和禁用钱包的 RPC 调用
-keypool = <n>	设置私钥池大小为 <n>（默认为 100）
-fallbackfee = <amt>	当交易估算没有足够数据时，该交易费（BTC/kB）将被使用（默认为 0.0002）
-mintxfee = <amt>	交易创建时，小于该交易费（BTC/kB）的被认为零交易费（默认为 0.00001）
-paytxfee = <amt>	发送的交易每 KB 字节的手续费（BTC/kB)(默认为 0.00）
-rescan	重新扫描区块链以查找钱包丢失的交易
-salvagewallet	启动时尝试从破坏的钱包文件 wallet.dat 恢复私钥
-sendfreetransactions	发送时尽可能不支付交易费用（默认为 0）
-spendzeroconfchange	付款时允许使用未确认的零钱（默认为 1）
-txconfirmtarget = <n>	如果未设置交易费用，则自动添加足够的交易费以确保交易在平均 n 个数据块内被确认（默认为 2）
-maxtxfee = <amt>	最大单次转账费用（BTC），设置太低可能会导致大宗交易失败（默认为 0.10）

（续）

比特币核心版本 v0.12.1 使用：bitcoin-qt 或 bitcoind [命令行选项]	
-upgradewallet	程序启动时升级钱包到最新格式
-wallet = <file>	指定钱包文件（数据目录内）(默认为 wallet.dat)
-walletbroadcast	钱包广播事务处理（默认为 1）
-walletnotify = <cmd>	当最佳区块变化时执行命令（命令行中的 %s 会被替换成区块 Hash 值）
-zapwallettxes = <mode>	删除钱包的所有交易记录，且只有用 -rescan 参数启动客户端才能重新取回交易记录（1 = 保留交易元数据，如账户所有者和支付请求信息，2 = 不保留交易元数据）
ZeroMQ 通知选项：	
-zmqpubhashblock = <address>	允许在 <address> 广播 Hash 区块
-zmqpubhashtx = <address>	允许在 <address> 广播 Hash 交易
-zmqpubrawblock = <address>	允许在 <address> 广播原始区块
-zmqpubrawtx = <address>	允许在 <address> 广播原始交易
调试 / 测试选项：	
-uacomment = <cmt>	附加注释到 User Agent 字符串
-debug = <category>	输出调试信息（默认为 0，提供 <category> 是可选项）。如果 <category> 未提供或 <category> = 1，则输出所有调试信息。<category> 可能是：addrman、alert、bench、coindb、db、lock、rand、rpc、selectcoins、mempool、mempoolrej、net、proxy、prune、http、libevent、tor、zmq 或 qt
-gen	生成比特币（默认为 0）
-genproclimit = <n>	设置比特币生成线程数（–1 = 所有核，默认为 1）
-help-debug	显示所有调试选项
-logips	在调试输出中包含 IP 地址（默认为 0）
-logtimestamps	输出调试信息时，前面加上时间戳（默认为 1）
-minrelaytxfee = <amt>	当转发、挖矿和交易创建时，小于该设置的交易费（BTC/kB）被认为是 0（默认为 0.00001）
-printtoconsole	跟踪 / 调试信息输出到控制台，不输出到 debug.log 文件
-shrinkdebugfile	客户端启动时压缩 debug.log 文件（默认 no-debug 模式时为 1）
区块链网络选项：	
-testnet	在测试网络中运行，而不是在真正的比特币网络中
节点中继选项：	
-bytespersigop	在中继和挖矿时，交易中每个 sigop 的最小字节数（默认为 20）
-datacarrier	是否接受中继和挖矿的带外交易（默认为 1）
-datacarriersize	中继和挖矿的带外交易数据最大值（默认为 83，单位为字节）
-mempoolreplacement	启用内存池交易替换（默认为 1）
数据块创建选项：	
-blockminsize = <n>	设置最小区块大小（默认为 0，单位为字节）
-blockmaxsize = <n>	设置最大区块大小（默认为 750000，单位为字节）
-blockprioritysize = <n>	设置高优先级 / 低交易费交易的最大字节（默认为 0）
RPC 服务器选项：	
-server	接受命令行和 JSON-RPC 命令

（续）

比特币核心版本 v0.12.1 使用：bitcoin-qt 或 bitcoind［命令行选项］	
-rest	接受公共 REST 请求（默认为 0）
-rpcbind = <addr>	绑定到指定地址监听 JSON-RPC 连接。IPv6 使用［主机］：端口格式。该选项可多次指定（默认为绑定到所有接口）
-rpccookiefile = <loc>	验证 cookie 的位置（默认为数据目录）
-rpcuser = <user>	JSON-RPC 连接时用的用户名
-rpcpassword = <pw>	JSON-RPC 连接时用的密码
-rpcauth = <userpw>	JSON-RPC 连接时用的用户名和 Hash 密码。<userpw> 格式：<USERNAME>:<SALT>$<HASH>。目录 share/rpcuser 下有一个权威的 Python 脚本可以使用。这个选项可以配置多次。
-rpcport = <port>	使用 <port> 端口监听 JSON-RPC 连接（默认为 8332；testnet：18332）
-rpcallowip = <ip>	允许来自指定地址的 JSON-RPC 连接。<ip> 为单一 IP（如：1.2.3.4），网络 / 掩码（如：1.2.3.4/255.255.255.0），网络 /CIDR（如：1.2.3.4/24），该选项可以多次指定
-rpcthreads = <n>	设置 RPC 服务线程数（默认为 4）
界面选项：	
-choosedatadir	在启动时选择目录（默认为 0）
-lang = <lang>	设置语言，例如“zh-CN”（默认为系统语言）
-min	启动时最小化
-rootcertificates = <file>	设置付款请求的 SSL 根证书（默认为 - 系统 -）
-splash	显示启动画面（默认为 1）
-resetguisettings	重置所有图形界面所做的更改

4.4.2 RPC API 调用接口

bitcoind 除了提供上节的命令行调用接口，还提供了 RPC API 调用接口以方便通过程序代码调用，如表 4-26 所示。

表 4-26 RPC 命令及说明

RPC 命令	解释说明
区块链模块	
getbestblockhash	获取主链中高度最大的区块的 Hash
getblock "hash" (verbose)	根据指定的索引，返回对应的区块信息
getblockchaininfo	获取区块链信息
getblockcount	获取主链中区块的数量
getblockhash index	根据指定的索引，返回对应区块的 Hash 值
getblockheader "hash" (verbose)	根据指定的索引，返回对应区块的头部信息
getchaintips	获取包括分叉链在内的所有区块链的最大区块信息
getdifficulty	获取挖矿难度
getmempoolancestors txid (verbose)	获取内存池对应 Hash 的信息，正序排列
getmempooldescendants txid (verbose)	获取内存池对应 Hash 的信息，逆序排列

（续）

RPC 命令	解释说明
getmempoolentry txid	返回指定交易的内存数据
getmempoolinfo	返回内存池信息
getrawmempool (verbose)	获取内存中未确认的交易列表
gettxout "txid" n (includemempool)	根据指定的 Hash 和索引，返回对应的零钱信息
gettxoutproof ["txid", ...] (blockhash)	返回某个 txid 在某个块的证据
gettxoutsetinfo	获取已确认的未支付交易的统计信息
verifychain (checklevel numblocks)	验证区块链数据库
verifytxoutproof "proof"	验证 gettxoutproof 返回的证据
控制模块	
getinfo	获取统计信息
help ("command")	帮助
stop	退出程序
创建模块	
generate numblocks (maxtries)	立即生成 x 个块（仅用于回归测试模式）
generatetoaddress numblocks address (maxtries)	立即生成 x 个块并发向地址 Y（仅用于回归测试模式）
挖矿模块	
getblocktemplate ("jsonrequestobject")	获取挖矿模板
getmininginfo	获取挖矿信息
getnetworkhashps (blocks height)	获取估算的挖矿 Hash 算力（hashes per second）
prioritisetransaction <txid><priority delta><fee delta>	提高挖矿时的交易被打包的优先级
submitblock "hexdata" ("jsonparametersobject")	提交广播新块到网络
网络模块	
addnode "node""add|remove|onetry"	尝试从 addnode 列表加入或删除节点，或者尝试连接节点
clearbanned	清理被禁的 IPs
disconnectnode "node"	立刻从指定节点断开
getaddednodeinfo dummy ("node")	获取节点信息
getconnectioncount	获取节点当前的连接数
getnettotals	获取网络流量统计信息
getnetworkinfo	获取网络信息
getpeerinfo	获取连接上的节点信息
listbanned	列出所有被禁用的 IPs
ping	发送 ping 命令
setban "ip(/netmask)""add|remove" (bantime) (absolute)	尝试从禁用列表加入或删除节点
交易模块	
createrawtransaction [{"txid":"id", "vout":n}, ...] {"address":amount, "data":"hex", ...} (locktime)	创建交易
decoderawtransaction "hexstring"	解码交易
decodescript "hex"	解码脚本

（续）

RPC 命令	解释说明
fundrawtransaction "hexstring" (options)	向 createrawtransaction 创建的交易里添加 input 直到满足 amount
getrawtransaction "txid" (verbose)	根据指定的 Hash 值，返回对应的交易信息
sendrawtransaction "hexstring" (allowhighfees)	广播交易
signrawtransaction "hexstring" ([{"txid":"id", "vout":n, "scriptPubKey":"hex", "redeemScript":"hex"}, ...] ["privatekey1", ...]sighashtype)	签名交易
工具模块	
createmultisig nrequired ["key", ...]	创建多签地址
createwitnessaddress "script"	创建隔离认证地址
estimatefee nblocks	评估达到 n 个块确认的交易费
estimatepriority nblocks	评估优先级
signmessagewithprivkey "privkey""message"	用私钥签名消息
validateaddress "bitcoinaddress"	获取比特币地址信息
verifymessage "bitcoinaddress""signature""message"	用比特币地址（公钥）验证消息
钱包模块	
abandontransaction "txid"	启用交易，从而使其输入再次变的可用
addmultisigaddress nrequired ["key", ...] ("account")	添加多签地址
addwitnessaddress "address"	添加隔离认证地址
backupwallet "destination"	备份 wallet.dat 钱包文件，恢复的时候可以通过 importwallet 进行恢复
dumpprivkey "bitcoinaddress"	打印地址私钥
dumpwallet "filename"	dump 钱包成可读文件的形式
encryptwallet "passphrase"	加密钱包
getbalance ("account" minconf includeWatchonly)	获取余额
getnewaddress ("account")	生成一个新的地址
getrawchangeaddress	生成一个找零地址
getreceivedbyaddress "bitcoinaddress" (minconf)	获取某个地址上接收的金额
gettransaction "txid" (includeWatchonly)	获取钱包里某笔交易的详细信息
getunconfirmedbalance	获取未确认的余额
getwalletinfo	获取钱包信息
importaddress "address" ("label" rescan p2sh)	导入地址
importprivkey "bitcoinprivkey" ("label" rescan)	导入私钥
importprunedfunds	导入资金
importpubkey "pubkey" ("label" rescan)	导入公钥
importwallet "filename"	恢复 backupwallet 命令备份的钱包
keypoolrefill (newsize)	预先生成地址
listaccounts (minconf includeWatchonly)	列出账号列表
listaddressgroupings	列出地址组
listlockunspent	列出临时未支付的输出

（续）

RPC 命令	解释说明
listreceivedbyaddress (minconf includeempty includeWatchonly)	列出地址列表余额
listsinceblock ("blockhash" target-confirmations includeWatchonly)	列出自某个区块以来的所有交易
listtransactions ("account" count from includeWatchonly)	列出一段区间之内的交易
listunspent (minconf maxconf ["address", ...])	列出未使用的交易
lockunspent unlock ([{"txid":"txid", "vout":n}, ...])	锁定或解锁交易
removeprunedfunds "txid"	从钱包删除指定交易
sendmany "fromaccount" {"address":amount, ...} (minconf "comment" ["address", ...])	向多个地址同时发币
sendtoaddress "bitcoinaddress" amount ("comment" "comment-to" subtractfeefromamount)	向 1 个地址同时发币
settxfee amount	设置交易费，覆盖 paytxfee 参数
signmessage "bitcoinaddress""message"	用指定地址的私钥签名消息

4.4.3　如何调用 API 进行开发

RPC API 可以通过多种方式调用开发，下面以 getbestblockhash 为例进行说明。

（1）通过 bitcoind-qt 调用

可以通过点击“帮助→调试窗口→控制台”进入 Bitcoin Core 的 RPC 控制台，输入 help 可以浏览所有的 RPC 命令，help getbestblockhash 可以查看 getbestblockhash 命令的详细帮助，包括输入、输出及说明。

（2）通过 bitcoind-cli 调用

先运行 bitcoind，然后运行 bitcoin-cli getbestblockhash。

（3）通过 curl 调用

调用代码如下：

```
curl --user myusername --data-binary '{"jsonrpc": "1.0", "id":"curltest",
"method": "getbestblockhash", "params": [] }' -H 'content-type: text/plain;' http://
127.0.0.1:8332/
```

（4）通过语言库调用

python 库的地址为：https://github.com/jgarzik/python-bitcoinrpc。

更多其他语言库参照 https://en.bitcoin.it/wiki/API_reference_(JSON-RPC)。

4.4.4　通过命令实现区块链的查询实例

1）根据高度 height 查询 block hash：

```
./bitcoin-cli getblockhash 19999 00000000ba36eb929dc90170a96ee3efb76cbebee0e0e5
c4da9eb0b6e74d9124
```

2）然后根据 block hash 查询 block 信息：

```
./bitcoin-cli getblock 00000000ba36eb929dc90170a96ee3efb76cbebee0e0e5c4da9eb0b6
e74d9124
{
  "hash" : "00000000ba36eb929dc90170a96ee3efb76cbebee0e0e5c4da9eb0b6e74d9124",
  "confirmations" : 263032,
  "size" : 215,
  "height" : 19999,
  "version" : 1,
  "merkleroot" : "c1b09fa6bdc0b12b15cc1400d598ffed29dd33b2e282093a48646d1b7b380c98",
  "tx" : [
    "c1b09fa6bdc0b12b15cc1400d598ffed29dd33b2e282093a48646d1b7b380c98"
  ],
  "time" : 1248291140,
  "nonce" : 1085206531,
  "bits" : "1d00ffff",
  "difficulty" : 1.00000000,
  "chainwork" : "0000000000000000000000000000000000000000000000004e204e204e20",
  "previousblockhash" : "000000006eb5c2799b0f5fafab6435daeecef8e7f609b731c9879c
3f74f28c73",
  "nextblockhash" : "00000000770ebe897270ca5f6d539d8afb4ea4f4e757761a34ca82e172
07d886"
}
```

3）根据交易 ID 查询单笔交易的信息。

一般情况下，通过以下命令只能查询自己钱包的信息，若不是自己钱包的交易，则返回如下代码：

```
./bitcoin-cli getrawtransaction c1b09fa6bdc0b12b15cc1400d598ffed29dd33b2e282093
a48646d1b7b380c98
error: {"code":-5,"message":"Invalid or non-wallet transaction id"}
```

那该怎么办呢？下面直接分析代码查找原因：

```
// Return transaction in tx, and if it was found inside a block, its hash is placed in
hashBlock
bool GetTransaction(const uint256 &hash, CTransaction &txOut, uint256 &hashBlock,
bool fAllowSlow)
{
  CBlockIndex *pindexSlow = NULL;
  {
    LOCK(cs_main);
    {
      if (mempool.lookup(hash, txOut))
      {
        return true;
```

```
        }
      }

      if (fTxIndex) {
        CDiskTxPos postx;
        if (pblocktree->ReadTxIndex(hash, postx)) {
          CAutoFile file(OpenBlockFile(postx, true), SER_DISK, CLIENT_VERSION);
          CBlockHeader header;
          try {
            file >> header;
            fseek(file, postx.nTxOffset, SEEK_CUR);
            file >> txOut;
          } catch (std::exception &e) {
            return error("%s : Deserialize or I/O error - %s", __func__, e.what());
          }
          hashBlock = header.GetHash();
          if (txOut.GetHash() != hash)
            return error("%s : txid mismatch", __func__);
          return true;
        }
      }

      if (fAllowSlow) { //use coin database to locate block that contains transaction,
and scan it
        int nHeight = -1;
        {
          CCoinsViewCache &view = *pcoinsTip;
          CCoins coins;
          if (view.GetCoins(hash, coins))
            nHeight = coins.nHeight;
        }
        if (nHeight > 0)
          pindexSlow = chainActive[nHeight];
      }
    }

    if (pindexSlow) {
      CBlock block;
      if (ReadBlockFromDisk(block, pindexSlow)) {
        BOOST_FOREACH(const CTransaction &tx, block.vtx) {
          if (tx.GetHash() == hash) {
            txOut = tx;
            hashBlock = pindexSlow->GetBlockHash();
            return true;
          }
        }
      }
    }

    return false;
  }
```

可以看出，若 fTxIndex 为 true，则可以直接搜索 index 获取 block 信息。

通过 -reindex-txindex 建立索引，这两个参数的意义如下：

```
-txindex        Maintain a full transaction index (default: 0)
-reindex        Rebuild block chain index from current blk000??.dat files
```

用 bitcoind 运行这两个参数来建立索引：

```
./bitcoind -reindex -txindex
```

这个建立索引的过程在笔者的 Mac 上运行了数个小时，再次查询，结果如下：

```
./bitcoin-cli getrawtransaction c1b09fa6bdc0b12b15cc1400d598ffed29dd33b2e282093
a48646d1b7b380c98

01000000010000000000000000000000000000000000000000000000000000000000000000ffffff
fff0704ffff001d0168ffffffff0100f2052a01000000434104cab13751ae7f0e0e49f8fb345e931bc6
a6349502da0cbcad98e9d95110ebde5ca7af9eb09639c022ac251b44d0fa200b54011198a405984a8ff
92ea9028d6d60ac00000000
```

这时就可以查询到其他交易的十六进制信息，解码十六进制交易数据就可以看到 JSON 格式的具体信息：

```
./bitcoin-cli decoderawtransaction 01000000010000000000000000000000000000000000000000
0000000000000000000000000000000ffffffff0704ffff001d0168ffffffff0100f2052a01000000434
104cab13751ae7f0e0e49f8fb345e931bc6a6349502da0cbcad98e9d95110ebde5ca7af9eb09639c022
ac251b44d0fa200b54011198a405984a8ff92ea9028d6d60ac00000000
{
  "txid" : "c1b09fa6bdc0b12b15cc1400d598ffed29dd33b2e282093a48646d1b7b380c98",
  "version" : 1,
  "locktime" : 0,
  "vin" : [
    {
      "coinbase" : "04ffff001d0168",
      "sequence" : 4294967295
    }
  ],
  "vout" : [
    {
      "value" : 50.00000000,
      "n" : 0,
      "scriptPubKey" : {
        "asm" : "04cab13751ae7f0e0e49f8fb345e931bc6a6349502da0cbcad98e9d95110eb
de5ca7af9eb09639c022ac251b44d0fa200b54011198a405984a8ff92ea9028d6d60 OP_CHECKSIG",
        "hex" : "4104cab13751ae7f0e0e49f8fb345e931bc6a6349502da0cbcad98e9d95110
ebde5ca7af9eb09639c022ac251b44d0fa200b54011198a405984a8ff92ea9028d6d60ac",
        "reqSigs" : 1,
        "type" : "pubkey",
        "addresses" : [
          "1KWj3Jk8xvS6fDdhQBsfmerscSGsS6CMiS"
```

```
        }
      }
    }
```

以上过程基本满足了大部分查询需求：输入交易 ID、区块高度、Hash 值等，至于通过“地址”查询，需要搜集这个地址的历史交易的输入输出，并且存入数据库，方可进行方便的查询。以下是把历史交易存入数据库的部分代码，详情请看笔者参与的基于 PostgreSQL 的区块浏览器开源实现：http://github.com/haobtc/openblockchain。

```
function updateKeys($hash160,$pubkey,$blockhash)
{
  global $db;
  $address=hash160ToAddress($hash160);
   $result=pg_fetch_assoc(pg_query_params($db,"SELECT pubkey,encode(hash160,
'hex') AS hash160 FROM keys WHERE hash160=decode($1,'hex')",array($hash160)));
  if(!$result && !is_null($pubkey))
  {
    pg_query_params($db, "INSERT INTO keys VALUES (decode($1,'hex'),$2,decode($
3,'hex'),decode($4,'hex'));",array($hash160,$address,$pubkey,$blockhash));
  }
  else if(!$result)
  {
    pg_query_params($db, "INSERT INTO keys(hash160,address,firstseen) VALUES (decode
($1,'hex'),$2,decode($3,'hex'));",array($hash160,$address,$blockhash));
  }
  else if($result && !is_null($pubkey) && is_null($result["pubkey"]))
  {
    if($result["hash160"]!=strtolower(hash160($pubkey)))
    {
      sleep(10);
      die("Hashes don't match");
    }
    pg_query_params($db, "UPDATE keys SET pubkey = decode($1,'hex') WHERE hash1
60=decode($2,'hex');",array($pubkey,$hash160));
  }
}
```

4）如何获取一笔交易的输入地址？（此案例可用在获取打款地址上。）

如何根据 txid 获取打款地址呢？其实是可以基于 blockchain 的链式结构逆向推导。

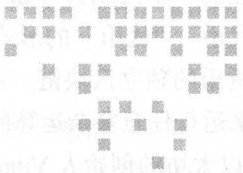
首先获得交易的 ID 号，交易 ID 号在交易输入 vin 中，以下 JSON 结构的 vin 中的 txid 后的一长串字符就是交易 ID 号，即 txid：

```
[vin] => Array
(
  [0] => Array
    (
      [txid] => 63876d10a13f3810a1d568c6ac7154f9b8a590cfc91cf8a17756fb099addf2b5
      [vout] => 1
```

```
        }
    }
```

根据txid的输出索引vout，在以下vout的JSON数据结构中，得到输入地址集addresses，其中的第0个地址就是我们想要的比特币地址：

```
[vout] => Array
    (
        [0] => Array
            (
                [value] => 0.16928006
                [n] => 0
                [scriptPubKey] => Array
                    (
                        [asm] => OP_DUP OP_HASH160 1715447427ac1cdfb7c5ba359154c37c5e9caa2b
OP_EQUALVERIFY OP_CHECKSIG
                        [hex] => 76a9141715447427ac1cdfb7c5ba359154c37c5e9caa2b88ac
                        [reqSigs] => 1
                        [type] => pubkeyhash
                        [addresses] => Array
                            (
                                [0] => LML1HJvP8jfeiwgSVmYfNGsedYfDrzKmq3
                            )
                    )
```



第 5 章 Chapter 5

以太坊智能合约开发

5.1 以太坊

5.1.1 以太坊的定义

以太坊是一个全新的开放的区块链平台，它允许任何人在平台中建立和使用通过区块链技术运行的去中心化应用。就像比特币一样，以太坊不受任何人控制，也不归任何人所有——它是一个开放源代码的项目，由全球范围内的很多人共同创建。和比特币协议有所不同的是，以太坊的设计十分灵活，极具适应性。在以太坊平台上创立新的应用十分便捷，随着 Homestead 的发布，任何人都可以安全地使用该平台上的应用。

5.1.2 下一代区块链

区块链技术是比特币的底层技术，这一技术第一次被描述是在中本聪 2008 年发表的白皮书《比特币：点对点电子现金系统》中。区块链技术更多的一般性用途在原书中已经有所讨论，但直到几年后，区块链技术才作为通用术语出现。一个区块链是一个分布式计算架构，里面的每个网络节点执行并记录着相同的交易，交易被分组为区块。一次只能增加一个区块，每个区块有一个数学证明来保证新的区块与之前的区块保持先后顺序。这样一来，区块链的“分布式数据库”就能与整个网络保持一致。个体用户与总账的互动（交易）将受到安全的密码保护。由数学执行并编码到协议中的经济激励因素刺激着维持和验证网络的节点。

在比特币中，分布式数据库被设想为一个账户余额表，一个总账，交易就是通过比特币的转移来实现个体之间无须信任基础的金融活动的。但是随着比特币吸引了越来越多开发者

和技术专家的注意，新的项目开始将比特币网络用于有价代币转移之外的其他用途。其中很多都采用了“代币”“代币”的形式——以原始比特币协议为基础，增加了新的特征或功能，采用各自加密货币的独立区块链。在 2013 年年末，以太坊的发明者 Vitalik Buterin 建议能够通过程序重组来运行任意复杂运算的单个区块链应该包含其他的程序。

2014 年，以太坊的创始人 Vitalik Buterin、Gavin Wood 和 Jeffrey Wilcke 开始研究新一代区块链，试图实现一个总体上完全无需信任基础的智能合约平台。

5.1.3 以太坊虚拟机

以太坊是可编程的区块链。它并不会给用户一系列预先设定好的操作（例如比特币交易），而是允许用户按照自己的意愿创建复杂的操作。这样一来，它就可以作为多种类型去中心化区块链应用的平台，包括加密货币在内，但并不仅限于此。

以太坊狭义上是指一系列定义去中心化应用平台的协议，它的核心是以太坊虚拟机（EVM），其可以执行任意复杂算法的编码。在计算机科学术语中，以太坊是“图灵完备的”。开发者能够使用以现有的 JavaScript 和 Python 等语言为模型的其他友好的编程语言，创建出在以太坊模拟机上运行的应用。

和其他区块链一样，以太坊也有一个点对点的网络协议。以太坊区块链数据库是由众多连接到网络的节点来维护和更新的。每个网络节点都运行着以太坊模拟机并执行相同的指令。因此，人们有时形象地称以太坊为“世界电脑”。

这个贯穿整个以太坊网络的大规模并行运算并不是为了使运算变得更加高效。实际上，这个过程会使得以太坊上的运算比在传统“电脑”上更慢更昂贵。然而，每个以太坊节点都运行着以太坊虚拟机是为了保持整个区块链的一致性。去中心化的一致性使得以太坊具有极高的故障容错性，保证零停机，而且还可以使存储在区块链上的数据永远保持不变且具有抗审查性。

以太坊作为一个智能合约平台，和编程语言相似，是由企业家和开发者来决定其用途的。不过很明显，某些应用类型较之更能从以太坊的功能中获益。以太坊尤其适合那些在点与点之间自动进行直接交互或跨网络促进小组协调活动的应用。例如，协调点对点市场的应用，或者是复杂财务合约的自动化。比特币使得个体能够不借助金融机构、银行或政府等其他中介来进行货币交换。以太坊的影响可能更为深远。理论上，任何复杂的金融活动或交易都能在以太坊上用编码自动且可靠地进行。除金融类应用之外，任何对信任、安全和持久性要求较高的应用场景——比如资产注册、投票、管理和物联网——都会大规模地受到以太坊平台的影响。

5.1.4 以太坊的工作原理

以太坊合并了很多对比特币用户来说十分熟悉的特征和技术，同时自己也进行了很多修正和创新。比特币区块链纯粹是一个关于交易的列表，而以太坊的基础单元是账户。以太坊

区块链跟踪每个账户的状态，以太坊区块链上的所有状态转换都是账户之间价值和信息的转移。这里所说的账户分为两类，具体如下。

- 外部账户（EOA）：由私人密钥控制。
- 合约账户：由它们的合约代码控制，只能由外部账户“激活”。

对于大部分用户来说，两者最基本的区别在于外部账户是由人类用户掌控的——因为他们能够控制私钥，进而控制外部账户。而合约账户则是由合约代码管控的。如果它们是被人类用户“控制”的，那也是因为程序设定它们可以被外部账户控制，进而被持有私钥的用户控制。“智能合约”这个流行的术语指的是在合约账户中编码——交易被发送给该账户时所运行的程序。用户可以通过在区块链中部署编码来创建新的合约。

只有当外部账户发出指令时，合约账户才会执行相应的操作。所以合约账户不可能自发地执行诸如任意数码生成或应用程序界面调用等操作——只有受外部账户调用时，它才会做这些事。这是因为以太坊要求节点能够与运算结果保持一致，这就要求保证正确执行所有操作。

和比特币一样，以太坊用户必须向网络支付少量的交易费用。这可以使以太坊区块链免受无关紧要或恶意的运算任务干扰，比如分布式拒绝服务（DDoS）攻击或无限循环。交易的发送者必须在激活的“程序”的每一步付款，包括运算和记忆储存。费用通过以太坊自有的有价代币、以太币的形式支付。

交易费用由节点收集，节点使网络生效。这些“矿工”就是以太坊网络中收集、传播、确认和执行交易的节点。矿工们将交易打包——包括许多以太坊区块链中账户“状态”的更新——打包成的数据被称为“区块”，矿工们会互相竞争，以使他们的区块可以添加到下一个区块链上。矿工们每挖到一个成功的区块就会得到以太币的奖励。这就为人们带来了经济激励，促使人们为以太坊网络贡献硬件和电力。

和比特币网络一样，矿工们通过解决复杂数学问题的任务以便成功地“挖”到区块，这被称为“工作量证明”。一个运算问题，如果在算法上解决，比验证解决方法需要更多数量级的资源，那么它就是工作量证明的极佳选择。为防止比特币网络中已经发生的专门硬件（例如特定用途集成电路）造成的中心化现象，以太坊选择了侧重于消耗更多内存的运算问题。如果问题需要内存和 CPU，那么理想的硬件就是普通的电脑。这就使以太坊的工作量证明具有抗特定用途集成电路的特性，和比特币这种由专门硬件控制挖矿的区块链相比，以太坊能够带来更加去中心化的安全分布。

5.2　以太坊账户管理

5.2.1　账户

账户在以太坊中发挥着中心作用。以太坊账户共有两种账户类型：外部账户（EOA）和

合约账户。这里我们重点讲一下外部账户，以下会简称为账户。合约账户简称为合约，在后面的 5.7 节会具体讨论合约的相关内容。把外部账户和合约账户都归入到账户这个概念里是合理的，因为这些实体都是所谓的状态对象。这些实体都有状态：账户有余额；合约既有余额也有合约数据。所有账户的状态代表的都是以太坊网络的状态，以太坊网络会和每一个区块一起更新，网络需要达成关于以太坊的共识。对于用户通过交易和以太坊区块链互动来说，账户是必不可少的。

如果把以太坊限制为只有外部账户，只允许外部账户之间进行交易，那么我们就会进入到“代币”系统，“代币”系统类似于比特币，只能用于转移以太币。

账户代表着外部代理人（例如人物角色、挖矿节点，或者是自动代理人）的身份。账户运用非对称公钥体制中的私钥来签署交易，以便以太坊虚拟机可以安全地验证交易发送者的身份。

5.2.2 钥匙文件

每个账户都由一对钥匙来定义，一个私钥和一个公钥。账户以地址为索引，地址由公钥衍生而来，取公钥的最后 20 个字节。每对私钥 / 地址都编码在一个钥匙文件里。钥匙文件是 JSON 文本文件，可以用任何文本编辑器打开和浏览。钥匙文件的关键部分——账户私钥，通常用你创建账户时设置的密码进行加密。可以在以太坊节点数据目录的 keystore 子目录下找到钥匙文件。请确保经常给钥匙文件备份！（查看 5.2 节账号备份和恢复可了解更多。）创建钥匙和创建账户一样具有如下特征。

- 不必告诉任何人你的操作。
- 不必和区块链同步。
- 不必运行客户端。
- 甚至不必连接到网络。

当然，新账户不包含任何以太币，它完全属于你。若没有你的钥匙和密码，没有人能够进入。

此外，转换整个目录或任何以太坊节点之间的个人钥匙文件都是安全的。

请注意，万一你从一个不同的节点向另一个节点添加钥匙文件，账户的顺序可能会发生改变。请确保不要回复或改变手稿中的索引或代码片段。

5.2.3 创建账号

为了从账户发送交易，包括发送以太币，你必须同时拥有钥匙文件和密码。确保钥匙文件有一个备份并牢记密码，然后尽可能安全地存储它们。如果钥匙文件丢失或忘记密码，就会丢失所有的以太币。没有密码不可能进入账号，也没有“忘记密码”的选项，所以一定不

要忘记密码。

记住密码并备份钥匙文件 <backup-and-restore-accounts>。

1. 使用 geth account new

geth 是以太坊的 go 语言客户端，一旦安装了 geth 客户端，创建账号就只是在终端执行 geth account new 指令了。注意，不必运行 geth 客户端，或者将其与区块链同步来使用 geth account 指令。geth account 指令的代码如下：

```
$ geth account new
Your new account is locked with a password. Please give a password. Do not
forget this password.
Passphrase:
Repeat Passphrase:
Address: {168bc315a2ee09042d83d7c5811b533620531f67}
```

对于非交互式使用，你可以提供纯文本密码文件作为 --password 标志的变元。文件中的数据包含密码的原始字节，后面可选择单独跟着新的一行，示例代码如下：

```
$ geth --password /path/to/password account new
```

用 --password 标志只是为了测试或在信任的环境中自动操作。不建议将密码保存在文件中或以任何其他的方式暴露在外。如果你用密码文件来使用 --password 标志，那么要确保文件只对你自己可阅读和列表。你可以在 Mac/Linux 系统中通过以下指令实现：

```
touch /path/to/password
chmod 600 /path/to/password
cat > /path/to/password
>I type my pass
```

要列出目前在你的 keystore 文件夹中的钥匙文件的所有账号，使用 geth account 指令的 list 子指令即可，示例代码如下：

```
$ geth account list
account #0: {a94f5374fce5edbc8e2a8697c15331677e6ebf0b}
account #1: {c385233b188811c9f355d4caec14df86d6248235}
account #2: {7f444580bfef4b9bc7e14eb7fb2a029336b07c9d}
```

钥匙文件的文件名格式为 UTC--<created_at UTC ISO8601>-<address hex>。账号列出时是按字母顺序排列的，但是由于时间戳格式，实际上它是按照创建顺序来排列的。

2. 使用 geth 控制台

为了使用 geth 创建新账号，我们必须先在控制台模式下开启 geth（或者可以用 geth

attach 将控制台依附在正在运行着的事例上），开启命令如下：

```
> geth console 2>> file_to_log_output
instance: Geth/v1.4.0-unstable/linux/go1.5.1
coinbase: coinbase: [object Object]
at block: 865174 (Mon, 18 Jan 2016 02:58:53 GMT)
datadir: /home/USERNAME/.ethereum
```

控制台使你能够通过发出指令与本地节点互相作用。比如，试一下这个列出账号的指令：

```
> eth.accounts
{
code: -32000,
message: "no keys in store"
}
```

这就表明你没有账号。你也可以从控制台创建一个账号：

```
> personal.newAccount()
Passphrase:
Repeat passphrase:
"0xb2f69ddf70297958e582a0cc98bce43294f1007d"
```

记得采用一个安全性强、随机生成的密码。

我们刚刚创建了第一个账号。如果我们再次试着列出账号，就可以看到新创建的账号了：

```
> eth.accounts
["0xb2f69ddf70297958e582a0cc98bce43294f1007d"]
```

3. 使用 Mist 以太坊钱包

对于相反的命令行，现在有一个基于 GUI 的选项可以用来创建账号：“官方”Mist 以太坊钱包。Mist 以太坊钱包和它的父项目 Mist，是在以太坊基金会的赞助下开发的，因此居于“官方”地位。钱包应用有 Linux、Mac OS X 和 Windows 可用的版本。

Mist 钱包是试用软件，使用风险需自行承担。

使用 GUI Mist 以太坊钱包创建账号再容易不过了。事实上，第一个账号在应用安装期间就创建出来了，安装步骤具体如下：

1）根据你的操作系统下载钱包应用的最新版本。由于你实际上会运行一个完整的 geth 节点，打开钱包应用就会开始同步复制你电脑上的整个以太坊区块链。

2）解压缩下载的文件夹，运行以太坊钱包的可执行文件。

3）等待区块链完全同步，按照屏幕上的说明进行操作，第一个账号就创建出来了。

第一次登录 Mist 以太坊钱包，你会看到自己在安装过程中创建的账号。它会被默认命名为主账号。

再另外创建账号也很容易；只需要点击应用主界面上的添加账号，输入所需的密码即可。

Mist 钱包仍在开发之中，以上列出的具体步骤可能会随着更新有所变更。

5.3　更新、备份、恢复账号

5.3.1　更新账号

你可以把钥匙文件更新到最新的钥匙文件格式，并且 / 或者升级钥匙文件密码。

在 geth 命令行中，可以用更新子命令来更新现在的账号，该过程可使用账号地址或索引作为参数。记住账号索引反映了创建顺序（按字母顺序排列的钥匙文件名包含了创建的时间）。示例代码如下：

```
geth account update b0047c606f3af7392e073ed13253f8f4710b08b6
```

或者：

```
geth account update 2
```

例如：

```
$ geth account update a94f5374fce5edbc8e2a8697c15331677e6ebf0b
Unlocking account a94f5374fce5edbc8e2a8697c15331677e6ebf0b | Attempt 1/3
Passphrase:
0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b
account 'a94f5374fce5edbc8e2a8697c15331677e6ebf0b' unlocked.
Please give a new password. Do not forget this password.
Passphrase:
Repeat Passphrase:
0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b
```

账户以加密的形式存储在最新版本中，它会提示你需要一个密码来解锁账户，另一个密码用来保存更新的文件。同一个指令还可以用于将弃用格式的账户变成最新版本，或者改变账户密码。

对于非交互式使用，密码可以用 --password 标志进行详细说明：

```
geth --password <passwordfile> account update a94f5374fce5edbc8e2a8697c15331677e6ebf0bs
```

由于只能给出一个密码，所以只能执行格式更新，修改密码只在交互式的情况下才有可能。

注意 账号更新有一个副作用那就是会引起账号顺序的变化。更新成功后，同一钥匙所有之前的格式 / 版本都会被移除！

5.3.2 账号备份和恢复

1. 手动备份 / 恢复

要从账号发送交易，需要有账号钥匙文件。钥匙文件可以在以太坊节点数据目录的 keystore 子目录下找到。数据目录的默认位置与平台相关，具体如下。

- Windows：在 C:\Users\username\%appdata%\Roaming\Ethereum\keystore 中。
- Linux：在 ~/.ethereum/keystore 中。
- Mac：在 ~/Library/Ethereum/keystore 中。

要备份钥匙文件（账号），则要在 keystore 子目录中复制单独的钥匙文件或复制整个 keystore 文件夹。

要恢复钥匙文件（账号），则要将钥匙文件重新复制到 keystore 子目录中，即其原始地址。

2. 导入未加密私钥

通过 geth 来导入未加密私钥，示例如下：

```
geth account import /path/to/<keyfile>
```

这个指令用于从纯文本文件 <keyfile> 导入未加密私钥并创建新的账号和打印地址。假定钥匙文件包含了未加密私钥作为编码到十六进制的标准 EC 原始字节。账号将以加密的形式储存，会提示你输入密码。你需要记住密码用于以后解锁账号。

下面给出一个例子，以详细说明数据目录。如果没有使用 --datadir 标志，那么就会在默认数据目录里创建新数据，例如钥匙文件会被放在数据目录的钥匙文件子目录里。

```
$ geth --datadir /someOtherEthDataDir account import ./key.prv
The new account will be encrypted with a passphrase.
Please enter a passphrase now.
Passphrase:
Repeat Passphrase:
Address: {7f444580bfef4b9bc7e14eb7fb2a029336b07c9d}
```

对于非交互式使用，密码可以用 --password 标志详细说明：

```
geth --password <passwordfile> account import <keyfile>
```

注意：因为你可以直接把 keystore 文件复制到另一个以太坊实例中，因此在节点之间转移账号的时候就不需要这个导入 / 导出机制了。

警告 当你往已存在节点的 keystore 里复制钥匙的时候，你所习惯的账户顺序可能会发生改变。因此要保证你不依赖于账户顺序，否则就要进行复核并更新脚本中使用的索引。

5.4　公有链、联盟链、私有链及网络配置

5.4.1　以太坊网络

去中心化共识的基础是参与节点的点对点网络，节点维持和保护着区块链（维护并保证区块链网络的安全）。

以太坊网络数据

EthStats.net 是以太坊网络实时数据的仪表板，这个仪表板展示了重要的信息，诸如现在的区块、散表难度、gas 价格和 gas 花费等。页面上显示的节点只是精选了网络上的实际节点。任何人都可以在 EthStats 仪表板上添加他们的节点。Github 上的 Eth-Netstats README 描述了如何连接。

EtherNodes.com 则展示了节点数的当前数据和历史数据，以及以太坊主网络和 Modern 测试网络上的其他信息。

在当前实时网络上，客户端实现分配 --EtherChain 上的实时数据。

5.4.2　公有链、私有链和联盟链

当今大多数以太坊项目都依靠以太坊作为公有链，公有链可以访问到更多的用户、网络节点、货币和市场。然而私有链或联盟链（在一群值得信任的参与者中）更适用于企业级区块链应用，例如，银行领域的很多公司都希望以太坊作为他们私有链的平台。

公有链、联盟链和私有链三种区块链在许可方面的区别如下。

- 公有链：世界上所有人都可以阅读和发送交易。如果他们是合法的，则都有希望看到自己被包括在内。世界上任何人都能参与到共识形成过程——决定在链条上添加什么区块及其现状是怎样的。作为中心化或准中心化信任的替代品，公有链受加密经济的保护，加密经济是经济激励和加密图形验证的结合，采用类似工作量证明或权益证明的机制，遵循的总原则是人们影响共识形成的程度和他们能够影响的经济资源数量成正比。这类区块链通常被认为是“完全去中心化的”。
- 联盟链：共识形成过程由预先选择的一系列的节点所掌控，例如，设想一个有 15 个金融机构的团体，每个机构都操作一个节点，为了使区块生效，其中的 10 个必须签署那个区块。阅读区块链的权利可能是公开的，或者仅限于参与者，也有混合的路径，比如将区块的根散表和应用程序编程接口一起公开，使公共成员可以进行一定量的查询，重获一部分区块链状态的加密图形证明。这类区块链被认为是“部分去中心化的。”
- 私有链：书写许可对一个组织保持中心化。阅读许可可能是公开的或是限制在任意程度。应用很可能包含对单个公司内部的数据库管理、审查等，因此公共的可读性在很多情况下根本是不必要的，但在另一些情况下人们又想要公共可读性。

私有链 / 联盟链与公有链可能毫无联系，但其属于以太坊区块链上的创新成果，对以太

坊整体生态系统有利。经过一段时间之后，这些会转变成软件改善、知识共享和工作机会。

5.4.3 如何连接

geth 会持续尝试在网络上连接到其他节点，直到有了端点为止。如果你在路由器上有可用的 UPnP，或者在面向因特网的服务器上运行以太坊，那么它也会接受其他节点的连接。

geth 可通过发现协议找到对等端。在发现协议中，节点互相闲聊以发现网络上的其他节点。最开始，geth 会使用一系列辅助程序节点，这些辅助程序节点的端点记录在源代码中。

检查连接和 ENODE 身份

要想检查客户端在交互控制台上连接了多少对等端点，net 模块有两个属性可以提供信息，告诉你对等端点的数量，以及你是否在监听的节点，示例代码如下：

```
> net.listening
true
> net.peerCount
4
```

要了解关于连接对等端点的更多信息，比如 IP 地址、端口号和支持协议，可使用管理员对象的 peers() 功能。admin.peers() 会返回到现在已连接的对等端点列表。示例代码如下：

```
> admin.peers
[{
ID: 'a4de274d3a159e10c2c9a68c326511236381b84c9ec52e72ad732eb0b2b1a2277938f78593
cdbe734e6002bf23114d434a085d260514ab336d4acdc312db671b',
Name: 'Geth/v0.9.14/linux/go1.4.2',
Caps: 'eth/60',
RemoteAddress: '5.9.150.40:30301',
LocalAddress: '192.168.0.28:39219'
}, {
ID: 'a979fb575495b8d6db44f750317d0f4622bf4c2aa3365d6af7c284339968eef29b69ad0dce
72a4d8db5ebb4968de0e3bec910127f134779fbcb0cb6d3331163c',
Name: 'Geth/v0.9.15/linux/go1.4.2',
Caps: 'eth/60',
RemoteAddress: '52.16.188.185:30303',
LocalAddress: '192.168.0.28:50995'
}, {
ID: 'f6ba1f1d9241d48138136ccf5baa6c2c8b008435a1c2bd009ca52fb8edbbc991eba36376be
aee9d45f16d5dcbf2ed0bc23006c505d57ffcf70921bd94aa7a172',
Name: 'pyethapp_dd52/v0.9.13/linux2/py2.7.9',
Caps: 'eth/60, p2p/3',
RemoteAddress: '144.76.62.101:30303',
LocalAddress: '192.168.0.28:40454'
}, {
ID: 'f4642fa65af50cfdea8fa7414a5def7bb7991478b768e296f5e4a54e8b995de102e0ceae2e
826f293c481b5325f89be6d207b003382e18a8ecba66fbaf6416c0',
Name: '++eth/Zeppelin/Rascal/v0.9.14/Release/Darwin/clang/int',
Caps: 'eth/60, shh/2',
```

```
RemoteAddress: '129.16.191.64:30303',
LocalAddress: '192.168.0.28:39705'
} ]
```

要检查 geth 的节点信息，可使用如下命令：

```
> admin.nodeInfo
{
Name: 'Geth/v0.9.14/darwin/go1.4.2',
NodeUrl: 'enode:// 3414c01c19aa75a34f2dbd2f8d0898dc79d6b219ad77f8155abf1a287ce2b
a60f14998a3a98c0cf14915eabfdacf914a92b27a01769de18fa2d049dbf4c17694@[::]:30303',
NodeID: '3414c01c19aa75a34f2dbd2f8d0898dc79d6b219ad77f8155abf1a287ce2ba60f14998
a3a98c0cf14915eabfdacf914a92b27a01769de18fa2d049dbf4c17694',
IP: '::',
DiscPort: 30303,
TCPPort: 30303,
Td: '2044952618444',
ListenAddr: '[::]:30303'
}
```

5.4.4　更快地下载区块链

启动以太坊客户端时，会自动下载以太坊区块链。用于下载以太坊区块链的时间会根据客户端、客户端设置、连接速度和可用的端点数量的变化而变化。下面是更快获取以太坊区块链的一些选项。

1. 使用 geth 下载

如果你正在使用 geth 客户端，那么可以做些什么来加速以太坊区块的下载速度呢？如果你用 --fast 标志来执行以太坊快速同步，那么就不会保留过去的交易数据。

你不能在执行所有或部分正常的同步操作之后再使用这个标志，也就是说在使用这个指令之前，不能下载以太坊区块链的任何部分。可查看这个 Ethereum Stack.Exchange answer 以了解更多。

下面是想要更快同步客户端时使用的一些标志。

```
--fast
```

这个标志使得通过状态下载而不是下载整个区块数据来实现快速同步成为可能。这样也能大幅减少区块链的尺寸。

--fast 只在从头开始同步区块链，并且会出于安全的原因在第一次下载区块链时，才会运行。

```
--cache=1024
```

分配到内部缓存的千兆内存（最少 16MB/ 数据库）。默认是 16MB，所以根据你电脑内存的大小，增加到 256、512、1024（1GB）或 2048（2GB）会有所不同。

```
--jitvm
```

这个标志可以激活 JIT VM。

完整的控制台命令示例如下：

```
geth --fast --cache=1024 --jitvm console
```

2. 导出 / 导入区块链

如果你已经同步了整个以太坊节点，那么你可以从完全同步的节点中导出区块链数据并将其导入新节点。你可以在 geth 中用 geth export filename 指令导出所有的节点，并用 geth import filename 将区块链导入节点来实现这一目的。

5.4.5 静态节点、信任节点和启动节点

geth 支持一个称为静态节点的特征，如果你有特定的端点，那么你会一直想与静态节点连接。如果断开连接，那么静态节点会再次连接。你可以配置永久性静态节点，方法是将如下代码放入 <datadir>/static-nodes.json 中（这应该是与 chaindata 及 keystone 处于同一个文件夹中）：

```
[
"enode:// f4642fa65af50cfdea8fa7414a5def7bb7991478b768e296f5e4a54e8b995de102e0ce
ae2e826f293c481b5325f89be6d207b003382e18a8ecba66fbaf6416c0@33.4.2.1:30303",
"enode://pubkey@ip:port"
]
```

你也可以在运行期间通过 JavaScript 使用 admin.addPeer() 加入静态节点，示例代码如下：

```
> admin.addPeer("enode:// f4642fa65af50cfdea8fa7414a5def7bb7991478b768e296f5e4a5
4e8b995de102e0ceae
```

连接的常见问题

有时候可能会无法连接，最常见的原因包括如下几点。

- 本地时间不正确。要想参与到以太坊网络中，需要有精确的时钟。检查 OS 如何同步时钟（例如 sudo ntpdate -s time.nist.gov），即便只快了 12 秒也有可能导致 0 端点。
- 有的防火墙配置可能会阻止 UDP 流通。可以利用静态节点功能或控制台上的 admin.addPeer() 来手动配置连接。

如果不想使用发现协议来启动 geth，也可以使用 --nodiscover 参数。不过，估计你只会在运行测试节点或有固定节点的实验测试网络时才想要这样做。

5.5　搭建测试网络和私有链

5.5.1　Modern 测试网

Modern 是公开的以太坊替代测试网。它会贯穿于整个软件里程碑 Frontier 和 Homestead 的生命周期。

1. 用法

eth(C++ 客户端)0.9.93 及以上版本自动支持 Modern 测试网。具体做法是：通过 --modern 参数开启以下任意客户端。

- PyEthApp（Python 客户端）：PyEthApp 支持 v1.0.5 以后的 Modern 网络。
- geth（Go 客户端）

2. 细节

除以下几条之外，Modern 测试网络的所有参数都与以太坊网络的相同。

- 网络名称：Modern
- 网络身份：2
- genesis.json
- 初始账户随机数（IAN）是 2^{20}（不像之前的网络中是 0）
  - 状态树形结构中的所有账户都有随机数 >= IAN。
  - 账户被插入到状态树形结构中时，都会被赋予一个初始随机数 = IAN。
- 初始通用区块散表：0cd786a2425d16f152c658316c423e6ce1181e15c3295826d7c9904cba9ce303
- 初始通用状态根：f3f4696bbf3b3b07775128eb7a3763279a394e382130f27c21e70233e04946a9

3. 获取测试网以太币

有两种方法可以获取测试网以太币，具体如下。

- 用 CPU/GPU 挖矿。
- 用以太坊 wei 龙头。

5.5.2　设置本地私有测试网

1. eth（C++ 客户端）

在该客户端，可以分别使用 --genesis 和 --config 连接或创建一个新的网络，也可以同时使用 --config 和 --genesis。只是，那样的话，--config 提供的初始区块描述会被 --genesis 选项覆盖。

本地私有测试网产生区块的方式，称为挖矿，以下是挖矿的相关参数。

- sealEngine：用来在区块中挖矿的引擎。
  - “Ethash”是以太坊工作量证明引擎（用于实时网络）。
  - “NoProof”在区块挖矿不需要工作量。
- params：是诸如 minGasLimit、minimumDifficulty、blockReward、networkID 等一般的网络信息。
- genesis：初始区块描述。
- accounts：设置包含账户 / 合约的初始状态。

内容与“--config”参数提供的初始领域相同。

2. geth（Go 客户端）

你可以在私有测试网上生成或挖掘自己的以太币。用这个方法试验以太坊很划算，可以避免不得不在公有链上挖矿，或去寻找测试网络的以太币。

在私有链中需要详细说明的事件具体如下。

- 定制初始文件。
- 定制数据目录。
- 定制网络 ID。
- （推荐）关闭节点发现。

3. 初始区块

初始区块是区块链的起始，即第一个区块，也称区块 0，它是唯一没有指向前面区块的一个区块。协议确保其他节点不会和你的区块链一致，除非他们和你拥有相同的初始区块，这样你想创建多少私有测试网区块链，就可以创建多少！示例如下：

```
{
"nonce": "0x0000000000000042", "timestamp": "0x0",
"parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
"extraData": "0x0", "gasLimit": "0x8000000", "difficulty": "0x400",
"mixhash": "0x0000000000000000000000000000000000000000000000000000000000000000",
"coinbase": "0x3333333333333333333333333333333333333333", "alloc": { }
}
```

初始区块的存储文件为 CustomGenesis.json。用下面的标志启动 geth 节点的时候，你会引用到这个。

```
--genesis /path/to/CustomGenesis.json
```

4. 私有网络的命令行参数

有一些必需的命令行选项（又称为“标志”）可以确保你的网络是私有的。前面已经谈到了初始标志，下面还有几个。注意下面的所有指令都会用在 geth 以太坊客户端。

```
--nodiscover
```

使用这个命令可以确保你的节点不会被非手动添加你的人发现。否则，你的节点可能会被陌生人的区块链无意添加，如果他和你拥有相同的初始文件和网络 ID 的话。

```
--maxpeers 0
```

如果你不希望其他人连接到你的测试链，那么可以使用 maxpeers 0。反之，如果你确切知道希望多少人连接到你的节点，那么你也可以通过调整数字来实现。

```
--rpc
```

这个指令可以激活你节点上的 RPC 界面。它在 geth 中默认是不激活的。

```
--rpcapi "db,eth,net,web3"
```

这个命令可以决定允许什么 API 能够通过 RPC 进入。在默认情况下，geth 可以在 RPC 激活 Web3 界面。

请注意在 RPC/IPC 界面提供 API，会使每个可以进入这个界面（例如 dapp's）的人都有权限访问这个 API。注意你激活的是哪个 API。geth 会默认激活 IPC 界面上所有的 API，以及 RPC 界面上的 db、eth、net 和 Web3 API。

```
--rpcport "8080"
```

将 8000 改变成你网络上开放的任何端口。geth 的默认设置是 8080。

```
--rpccorsdomain "http://chriseth.github.io/browser-solidity/"
```

这个可以指示什么 URL 能够连接到你的节点来执行 RPC 定制端任务。请务必谨慎，输入一个特定的 URL 而不是 wildcard (*)，后者会使所有的 URL 都能够连接到你的 RPC 实例。

```
--datadir "/home/TestChain1"
```

这是你的私有链数据所存储在的数据目录（在 nubits 下）。可以选择一个与你的以太坊公有链文件夹分开的位置。

```
--identity "TestnetMainNode"
```

这会为你的节点设置一个身份，使之更容易在端点列表中被辨认出来。这个例子说明了如何在网络上出现这些身份。

5. 发布 geth

待创建了定制初始区块 JSON 并建立了区块链数据目录之后，在控制台输入以下指令，即可进入 geth：

```
geth --identity "MyNodeName" --genesis /path/to/CustomGenesis.json --rpc --rpcport
"8080" --rpcco
```

请改变标志与定制设置匹配。

每次想要进入定制链的时候，你都需要使用定制链指令启动 geth 实例。如果你只在控制台输入“geth”，那么它将不会记住你设置的所有标志。

6. 给账户预分配以太币

“0x400”难度能让你在私有测试网链上快速挖出以太币。如果你创建了自己的链，开始挖矿，你应该在几分钟之内就会有上百个以太币，远远超过了在网络上测试交易所需的数量。如果你还想给账户预分配以太币，就需要按如下步骤进行。

1）创建私有链以后再创建新的以太坊账户。

2）复制新的账户地址。

3）在 Custom_Genesis.json 文件中添加以下指令：

```
"alloc":
{
"<your account address e.g. 0x1fb891f92eb557f4d688463d0d7c560552263b5a>":
{ "balance": "2000000000000000000" }
}
```

用你的账户地址取代 0x1fb891f92eb557f4d688463d0d7c560552263b5a

保存初始文件，重新运行私有链指令。完整装载 geth 以后，再关闭它。

如果想要指派一个地址给变量 primary，并查看它的余额，那么应该怎么操作呢？

在终端运行 geth account list 指令，查看指派给你的新地址账户号码是什么：

```
> geth account list
Account #0: {d1ade25ccd3d550a7eb532ac759cac7be09c2719}
Account #1: {da65665fc30803cb1fb7e6d86691e20b1826dee0}
Account #2: {e470b1a7d2c9c5c6f03bbaa8fa20db6d404a0c32}
Account #3: {f4dd5c3794f1fd0cdc0327a83aa472609c806e99}
```

记录你预分配以太币的账户号码。或者，用 geth console（和最先启动 geth 时保持一样的参数）启动控制台。提示出现以后，输入以下命令：

```
> eth.accounts
```

这会返回到你拥有的账户地址排列。

```
> primary = eth.accounts[0]
```

用你的账户指数取代 0，这个控制台指令会返回到你的第一个以太坊地址。

然后输入以下指令：

```
> balance = web3.fromWei(eth.getBalance(primary), "ether");
```

应该会返回到 7.5，这就意味着你账户里有那么多以太币。我们必须在你初始文件的分区里放入那么多数量，是因为“余额”领域以 wei 为单位取一个数字，wei 是以太坊货币以太币的最小面额。

```
https:// www.reddit.com/r/ethereum/comments/3kdnus/question_about_private_chain_mining_dont_upvote/
http:// adeduke.com/2015/08/how-to-create-a-private-ethereum-chain/
```

5.6　账户、交易核心概念及投注合约解析

5.6.1　外有账户与合约账户

前面说过，以太坊中有两种类型的账户，即外有（外部）账户和合约账户，它们的区别在 Serenity 版本中可能会消失。

1. 外有账户（EOA）

外有账户具有以下特征。

- ❑ 有以太币余额。
- ❑ 可以发送交易（以太币交易或引发合约代码）。
- ❑ 由私钥控制。
- ❑ 没有相关代码。

2. 合约账户

合约账户具有以下特征。

- ❑ 有以太币余额。
- ❑ 有相关代码。
- ❑ 代码执行由从其他合约接收的交易或信息（调用）触发。
- ❑ 执行的时候——执行任意复杂的操作（图灵完备的）——将会操控它自己的永久存储，例如，可以有自己的持久状态，还可以调用其他合约。

以太坊区块链上的所有行为都是由外有账户引发的交易调动。每次合约账户接收到交易时，它的代码都会按照输入参数的指示来执行，并作为交易的一部分来发送。合约代码由参与到网络的每个节点上的以太坊虚拟机来执行，作为验证新区块的一部分。

这个执行需要是完全确定性的，它唯一的语境是区块链上区块的位置和所有可见的数据。区块链上的区块代表时间单位，区块链本身是时间维度，代表在链上区块指定的离散时间点上状态的整个历史。

所有的以太币余额和价值都以 wei 为单位来命名：1 个以太币是 1e18 wei。

注意 不应该将以太坊中的“合约”看作是要“实现”或“遵守”的事物；它更像是在以太坊执行环境中生存的“自治代理”，当信息或交易“戳到”它的时候，总会执行特定的代码片段，并且对自己的以太币余额和钥匙 / 价值商店有直接的控制，以储存永久状态。

5.6.2 什么是交易

“交易”这个术语在以太坊中用来指代签署的数据包，数据包中存储着要从外有账户发送到区块链上另一账户的信息。

交易包括如下内容。

- 信息接收人。
- 一个签字，用于确认发送方身份，证明通过区块链向接收者发送信息的意图。
- VALUE 域，从发送方向接收方转移的 wei 的数量。
- 可选数据域，包括发送到合约的信息。
- 一个 STARTGAS 值，代表交易执行允许采取的运算步骤的最大数量。
- 一个 GASPRICE 值，代表发送人愿意支付的 gas 费用。一个 gas 单位对应着一个原子指令执行，比如运算步骤。

5.6.3 什么是消息

合约能够向其他合约发送“消息”。信息是虚拟的事物，永远不能序列化，只存在于以太坊执行环境中，它们可以被想象为功能调用。

消息具体包括以下内容。

- 消息发送方（内含的）。
- 消息接收方。
- VALUE 域，它和发送到合约地址的消息一起转移的 wei 的数量。
- 可选数据域，即发送到合约的实际数据。
- 一个 STARTGAS 值，限制了消息可以触发的代码执行的 gas 最大值。

本质上来说，一个消息就像一个交易，只不过消息是由合约而不是由外在因素创造的。当正在执行代码的合约执行 CALL 或 DELEGATECALL 操作码时，消息就产生了。和交易一样，消息可能会导致接收方账户运行代码。因此，就像和外在因素建立关系一样，合约也能以同样的方式和其他合约建立关系。

5.6.4 什么是 gas

以太坊虚拟机（EVM）是以太坊区块链上的可执行环境。每个参与到网络的节点都会运

行 EVM，以作为区块验证协议的一部分。他们检查列举在区块上的、他们验证的交易，运行 EVM 内部交易触发的代码。每个网络上的完整节点都进行同样的运算，存储相同的值。很明显，以太坊并不是关于运算效率的优化。与它类似的进程是多余的。它的目的是提供一个有效的方式，在系统状态上不需要信任的第三方、准则或暴力垄断就能达成共识。但重要的是他们不是为了优化运算而存在的。事实上，合约执行在节点之间被冗余地重复，自然使之运算成本更为昂贵，通常会鼓励人们如果能在链外操作运算，就不要在区块链里进行运算。

运行去中心化应用（dapp）时，它和区块链互动来阅读和修正状态，但是去中心化应用会很典型地只把业务逻辑和对共识至关重要的状态放在区块链上。

当信息或交易触发结果执行时，每个指令在每个网络节点被执行。这里会有一个代价：每个执行的操作都有特定的成本，其将以一定量的 gas 单元表现。

gas 是交易发送方需要为每个以太坊区块链上发生的操作所支付的执行花费。这个名字 gas 的灵感来自于这样一个观点，这笔花费就像加密燃料，驱使智能合约产生。gas 从执行代码的矿工处购买以获得以太币。gas 和以太币会被故意分开，因为 gas 单位与具备自然成本的运算单位一致，而以太币的价格通常会由于市场力量产生波动。二者由自由市场进行调和：gas 的价格实际上是由矿工决定的，矿工可以拒绝以低于最低限度的 gas 价格进行交易。为了获得 gas，你只需要向账户中添加以太币即可。以太坊客户端会自动为你的以太币购买 gas，数量是你指定的交易最大支出。

在合约或交易执行的每个运算步骤，以太坊协议都要收费，以防止以太坊网络上发生蓄意攻击或滥用的现象。每个交易都必须包含一个 gas 限度和每 gas 愿意支付的花费。矿工可以选择是否将交易包括在内和收集花费。如果交易产生的、用于运算步骤的 gas 总量，包括原始信息和可能引发的子信息，少于或等于 gas 限额，那么交易就会进行。如果 gas 总量超过 gas 限额，那么所有的变化都会复原，但是交易仍然有效，矿工仍然可以收集花费。未用于交易执行的、所有多余的 gas 都会以以太币的形式偿还给发送方。不必担心超支，因为只会对你消费的 gas 进行收费。这就意味着以高于预估的 gas 限额发送交易也是安全和有效的。

5.6.5　估算交易成本

交易花费的以太币总量基于以两个因素来计算。

- gasUsed：交易消费的 gas 总量。
- gasPrice：交易中指定的一个 gas 单元的价格（换算成以太币）。

总成本 = gasUsed * gasPrice

1. gasUsed

以太坊虚拟机上的每个操作都会被指派消费的 gas 数量。gasUsed 是所有执行的操作所需要的 gas 总额。

对于估算 gasUsed，可以用 estimate Gas API。

2. gasPrice

用户建构并签署交易，每个用户都可以说明自己想要的 gasPrice，其值可以是零。然而 Frontier 发布的以太坊客户端默认 gasPrice 是 0.05e12 wei。由于矿工会使收入最优化，因此，如果大部分交易都以 0.05e12 wei 的 gasPrice 提交，那么就很难说服矿工接受价格更低或为 0 的交易。

3. 示例交易成本

我们来做一个只添加 2 个数字的合约。EVM OPCODE ADD 消费 3 gas。大概的成本，以默认的 gas 价格来计算（2016 年 1 月）则是：

$$3 * 0.05e12 = 1.5e11 \text{ wei}$$

1 以太币是 1e18 wei，总成本就是 0.00000015 以太币。

这是个简化的计算，因为忽略了一些成本，比如将 2 个数字转移给合约的成本，在它们可以被添加之前。表 5-1 是各种常用合约操作的 gas 成本。

表 5-1 各种操作的 gas 成本

操作名称	gas 成本	备 注
step	1	每个执行循环的默认数量
stop	0	免费
suicide	0	免费
sha3	20	Hash
sload	20	移出永久存储
sstore	100	放进永久存储
balance	20	查询余额
create	100	合约创建
call	20	发起一个只读调用
memory	1	扩展内存时每个额外的词
txdata	5	一个交易的每个数据字节或编码
transaction	500	基础费用交易
contract creation	53000	在 homestead 从 21000 变化而来

5.6.6 账户交互示例：投注合约

之前提到过，账户的类型有两种，外有账户和合约账户。

以太坊默认的执行环境是没有生命的，什么都不会发生，每个账户的状态均保持相同。但是，每个用户都可以通过从外有账户发送交易来触发行动，启动以太坊。如果交易的目的地是其他外有账户，那么交易可能会转移一些以太币，否则什么也不会做。但如果目的地是个合约，那么合约会激活，自动运行代码。

代码有能力读 / 写自己的内部存储（一个将 32 字节钥匙映射到 32 字节价值的数据库），阅读存储的接收信息，给其他合约发送信息，转而触发执行。一旦执行停止，合约发送的信息所触发的所有的子执行都会停止（这些都将以决定好的同步的顺序来发生，比如，子调用在父调用进一步操作之前完全完成），执行环境再次立即停止，直到被下一个交易唤醒为止。

合约通常服务于四个目的，具体如下。

- 保持数据库代表着对其他合约或外部世界有用的东西；一个例子是合约激励货币，另一个例子是合约在特定的组织里记录会员。
- 作为某种具有更复杂访问政策的外有账户，其被称为“前向合约”，典型地只在特定条件下，把进来的信息转发给到指定目的地；例如，前向合约可能会等到指定 3 个私钥中的 2 个都确认了特定的信息之后才会进行转发（例如，多重签名）。更复杂的前向合约基于要发送的信息会有不同的条件。最简单的一个功能使用案例就是撤回限制，在一些更复杂的访问政策中难以驾驭。钱包合约就是一个很好的例子。
- 在多个用户之间管理一个正在进行的合约或关系。例子包括金融合约，由特定中介的第三方保管合约或一些保险。也可以是开放合约，一方对其他方的随时参与保持开放；一个例子是自动为提交数学问题有效解决方案或是证明提供了一些运算资源的人发奖金的合约。
- 给其他合约提供功能，本质上是作为软件库。

合约通过被交替称为“调用”或“发送信息”的活动进行互动。“信息”是包含一定量以太币，任何大小的数据字节串、发送方和接收方地址的事物。合约接收信息时，可以选择返还一些数据，信息本来的发送方可以立即使用。这样发送信息就和调用一个功能一样。

因为合约有这样的作用，我们期望合约可以彼此互动。举个例子，设想一个情景如图 5-1 所示，Alice 和 Bob 赌 100 Gav 币，明年旧金山的温度不会超过 35°C。但是 Alice 非常有安全意识，她的第一个账户使用的前向合约，只有在 3 个私钥中的 2 个都批准的情况下才可以发送信息。Bob 偏执于量子加密图形，他使用的前向合约，只传递有 Lamport 签名和传统 ECDSA 的信息（但是因为他很老派，所以更偏向于使用基于 SHA-256 的 Lamport 签名版本，以太坊不直接支持）。

投注合约本身需要从一些合约中取得旧金山天气的数据，当它想要实际发送 Gav 币给 Alice 或 Bob 时，也需要和 Gav 币合约交谈（或者，更准确地说，Alice 或 Bob 的前向合约）。因次我们可以像如下这样表示账户之间的关系。

Bob 想要最终决定赌注的时候，就会发生以下的步骤，具体如图 5-2 所示。

1）交易被发出，触发信息从 Bob 的外有账户发送到他的前向合约。

2）Bob 的前向合约给合约发送信息散表和 Lamport 签名，以发挥 Lamport 签名确认库的作用。

3）Lamport 签名确认库看到 Bob 想要基于 SHA-256 的 Lamport 签名，于是向 SHA-256 库多次发调用来确认签名。

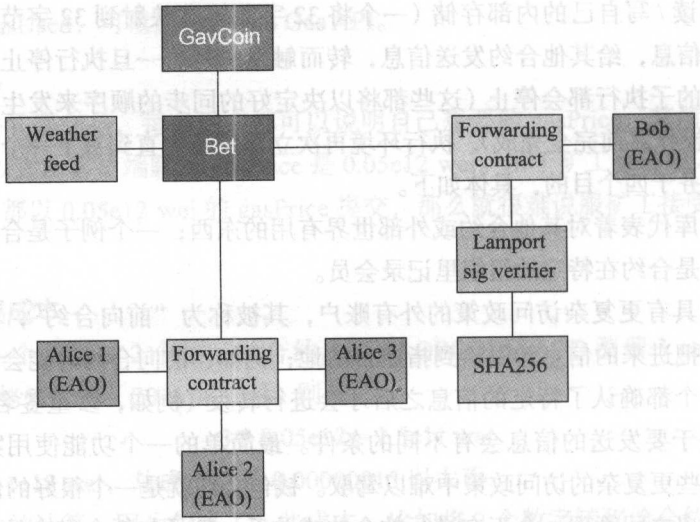


图 5-1 Gav 币的前向合约示意图

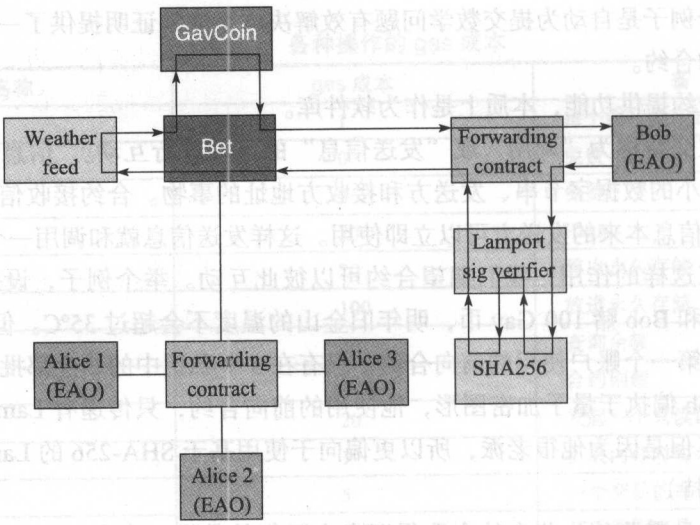


图 5-2 Gav 币合约的执行过程

4）一旦 Lamport 签名确认库回到 1，则表明签名已确认，他就会给代表赌注的合约发送信息。

5）赌注合约检查提供旧金山天气的合约，查看天气如何。

6）赌注合约看到对信息的回应显示天气高于 35°C，就会给 Gav 币合约发送信息，将 Gav 币从它的账户转移到 Bob 的前向合约。

注意，Gav 币在 Gav 币合约的数据库中是作为一个整体来“存储”的；第 6 步中“账户”的意思只是说在 Gav 币合约存储中有数据入口，有钥匙可以进入赌注合约的地址和余额值。

接收到信息后，Gav 币合约值会减少，与 Bob 前向账户对应的入口值会增加。

5.7　深入浅出智能合约

5.7.1　合约的定义

合约是代码（它的功能）和数据（它的状态）的集合，存在于以太坊区块链的特定地址。合约账户能够在彼此之间传递信息，进行图灵完备的运算。合约依靠被称作以太坊虚拟机（EVM）字节代码（以太坊特有的二进制格式）上的区块链来运行。

合约很典型地采用诸如 Solidity 等高级语言来写成，然后编译成字节代码上传到区块链上。

也有其他语言可以用于编写智能合约如 Serpent 和 LLL，这点将在 5.7.2 节做进一步阐述。去中心化应用开发资源列出了综合的开发环境，以帮助你用这些语言开发的开发者工具，提供测试和部署支持等功能。

5.7.2　以太坊高级语言

合约依靠被称作以太坊虚拟机（EVM）字节代码（以太坊特有的二进制格式）上的区块链来运行。然而，合约是典型地采用诸如 Solidity 等高级语言写成的，它会用以太坊虚拟机编译器编译成字节代码上传到区块链。

下面是开发者可以用来为以太坊编写智能合约的高级语言。

1. Solidity

Solidity 是一种与 JavaScript 相似的语言，你可以用它来开发合约并编译成以太坊虚拟机字节代码。目前它是以太坊最受欢迎的语言。

2. Serpent

Serpent 是一种与 Python 类似的语言，可以用于开发合约并编译成以太坊虚拟机字节代码。它力求简洁，将低级语言在效率方面的优点和操作简易的编程风格相结合，同时合约编程还增加了独特的领域特定功能。Serpent 用 LLL 进行编译。

3. LLL

Lisp Like Language (LLL) 是一种与 Assembly 类似的低级语言。它追求极简；本质上只是直接对以太坊虚拟机的一点包装。

4. Mutan（已弃用）

Mutan 是个静态类型，由 Jeffrey Wilcke 开发设计的 C 类语言。它已经不再受到维护。

5.7.3　写合约

没有 Hello World 程序，语言就不完整。Solidity 在以太坊环境内操作，没有明显的“输

出”字符串的方式。我们能做的最接近的事就是用日志记录事件来把字符串放进区块链，示例如下：

```
contract HelloWorld {
event Print(string out);
function() { Print("Hello, World!"); }
}
```

每次执行时，这个合约都会在区块链创建一个日志入口，打印“Hello,World!”参数。另请参阅：Solidity docs 里有更多关于写 Solidity 代码的示例和指导。

5.7.4 编译合约

Solidity 合约的编译可以通过很多机制来完成，具体如下。

- 通过命令行使用 solc 编译器实现。
- 在 geth 或 eth 提供的 JavaScript 控制台使用 web3.eth.compile.solidity（这里仍然需要安装 solc 编译器）实现。
- 通过在线 Solidity 实时编译器实现。
- 通过建立 Solidity 合约的 Meteor dapp Cosmo 实现。
- 通过 Mix IDE 实现。
- 通过以太坊钱包实现。

关于 solc 和编译 Solidity 合约代码的更多信息可在此查看。

1. 在 geth 中设置 Solidity 编译器

如果你启动了 geth 节点，就可以查看哪个编译器可用。示例代码如下：

```
> web3.eth.getCompilers();
["lll", "solidity", "serpent"]
```

这一指令会返回到显示当前哪个编译器可用的字符串。

solc 编译器与 cpp- ethereum 应一起安装。或者，你也可以自己创建。

如果你的 solc 可执行文件不在标准位置，那么可以用 --solc 标志为 solc 可执行文件指定一个定制路线，示例代码如下：

```
$ geth --solc /usr/local/bin/solc
```

或者你可以通过控制台在执行期间设置这个选项：

```
> admin.setSolc("/usr/local/bin/solc")
```

```
    solc, the solidity compiler commandline interface
    Version: 0.2.2-02bb315d/.-Darwin/appleclang/JIT linked to libethereum-1.2.0-8007
cef0/.-Darwin/appleclang/JIT
    path: /usr/local/bin/solc
```

2. 编译一个简单合约

让我们来编译一个简单的合约源，示例代码如下：

```
    > source = "contract test { function multiply(uint a) returns(uint d) { return a *
7; } }"
```

这个合约提供了一个单一方法 multiply，它和一个正整数 a 调用并返回到 a × 7。

下面准备在 geth JS 控制台用 eth.compile.solidity() 编译 Solidity 代码：

```
    > contract = eth.compile.solidity(source).test
    {
    code: '605280600c6000396000f3006000357c010000000000000000000000000000000000000000
00000000000000000000090048063c6888fa114602e57005b6037600436041565b80600052602060000f35
b60006007820290506004d565b91905056',
    info: {
    language: 'Solidity',
    languageVersion: '0',
    compilerVersion: '0.9.13',
    abiDefinition: [{
    constant: false,
    inputs: [{
    name: 'a',
    type: 'uint256'
    } ],
    name: 'multiply',
    outputs: [{
    name: 'd',
    type: 'uint256'
    } ],
    type: 'function'
    } ],
    userDoc: {
    methods: {
    }
    },
    developerDoc: {
    methods: {
    }
    },
    source: 'contract test { function multiply(uint a) returns(uint d) { return a
    *
    7; } }'
    }
    }
```

编译器能通过RPC因此也能通过Web3.js，对浏览器内任何可通过RPC/IPC连接到geth的Đapp都是可用的。

下面的例子将展示如何通过JSON-RPC接合geth来使用编译器：

```
    $ geth --datadir ~/eth/ --loglevel 6 --logtostderr=true --rpc --rpcport 8100
--rpccorsdomain ' * ' --mine console 2>> ~/eth/eth.log
    $ curl -X POST --data '{"jsonrpc":"2.0","method":"eth_compileSolidity","params"
:["contract test {
```

单源编译器输出会给出合约对象，每个都代表一个单独的合约。eth.compile.solidity的实际返还值是合约名字到合约对象的映射。由于合约名字是test，因此eth.compile.solidity(source).test会给出包含下列领域的测试合约对。

- Code：编译的以太坊虚拟机字节代码。
- Info：从编译器输出的额外元数据。
- Source：源代码。
- Language：合约语言（Solidity、Serpent、LLL）。
- LanguageVersion：合约语言版本。
- compilerVersion：用于编译这个合约的Solidity编译器版本。
- abiDefinition：应用的二进制界面定义。
- userDoc：用户的NatSpec Doc。
- developerDoc：开发者的NatSpec Doc。

编译器输出的直接结构化（到code和info）反映了两种非常不同的**部署**路径。编译的以太坊虚拟机代码和一个合约创建交易被发送到区块，剩下的（info）在理想状态下会存活在去中心化云上，公开验证的元数据则会执行区块链上的代码。

如果你的源包含多个合约，那么输出就会包括每个合约的一个入口，对应的合约信息对象可以用作为属性名称的合约名字检索到。你可以通过检测当前的GlobalRegistrar代码来试一下：

```
contracts = eth.compile.solidity(globalRegistrarSrc)
```

5.7.5 创建和部署合约

开始阅读5.7.5节之前，需要确保你有解锁的账户和一些资金。

现在在区块链上创建一个合约，方法是用5.7.4节的以太坊虚拟机代码作为数据给空地址发送交易，示例代码如下：

用在线Solidity实时编译器或Mix IDE程序会更容易完成。

```
    var primaryAddress = eth.accounts[0]
    var abi = [{ constant: false, inputs: [{ name: 'a', type: 'uint256' } ]
    var MyContract = eth.contract(abi)
    var contract = MyContract.new(arg1, arg2, ..., {from: primaryAddress, data:
evmByteCodeFromPrevio
```

所有的二进制数据都以十六进制的格式进行序列化。十六进制字符串总会有一个十六进制前缀 0x。

注意　arg1、arg2、……是合约构造函数参数，以备它要接受参数。如果合约不需要构造函数参数，则可以忽略这些参数。

值得注意的是，这一步需要你支付执行。一旦交易成功进入到区块，你的账户余额（你作为发送方放在 from 领域）就会根据以太坊虚拟机的 gas 规则被扣减。一段时间以后，你的交易会在一个区块中出现，确认它带来的状态是共识。你的合约现在存储在区块链上。

以不同步的方式做同样的事看起来就像下面这样：

```
    MyContract.new([arg1, arg2, ...,]{from: primaryAccount, data: evmCode},
function(err, contract) {
    if (!err && contract.address)
    console.log(contract.address);
    });
```

5.7.6　与合约互动

与合约互动的典型做法是用诸如 eth.contract() 功能的抽象层，它会返回到 JavaScript 对象，和所有可用的合约功能一起，作为可调用的 JavaScript 功能。

描述合约可用功能的标准方式是 ABI 定义。这个对象是一个字符串，它描述了调用签名和每个可用合约功能的返回值，示例代码如下：

```
    var Multiply7 = eth.contract(contract.info.abiDefinition);
    var myMultiply7 = Multiply7.at(address);
```

现在 ABI 中具体说明的所有功能调用都在合约实例中可用。你可以用两种方法中的一种来调用这些合约实例上的方法：

```
    > myMultiply7.multiply.sendTransaction(3, {from: address})
    "0x12345"
    > myMultiply7.multiply.call(3)
    21
```

当用 sendTransaction 调用的时候，通过发送交易来执行功能调用。需要花费以太币来进行发送，调用会永久记录在区块链上。用这种方式进行调用的返回值是交易散表。

当用 call 调用的时候，将在以太坊虚拟机中本地执行功能，功能返回值和功能一起返

回。用这种方式进行的调用不会记录在区块链上，因此也不会改变合约的内部状态。这种调用方式被称为**恒定**功能调用。以这种方式进行的调用不需要花费以太币。

如果你只对返回值感兴趣，那么你应该使用 call。如果你只关心合约状态的副作用，就应该使用 sendTransaction。

上面的例子不会产生副作用，因此 sendTransaction 只会烧 gas，增加宇宙的熵。

5.7.7 合约元数据

5.7.5 节已经介绍了如何在区块链上创建合约。现在就来处理剩下的编译器输出，合约元数据，或者说合约信息。

在与不是你创建的合约进行互动时，你可能会想要文档或是查看源代码。区块链鼓励合约作者提供这样的可见信息，他们可以在区块链上登记或借助第三方服务，比如说 EtherChain 来提供。管理员 API 为所有选择登记的合约提供便利的方法来获取这个捆绑，示例代码如下：

```
//get the contract info for contract address to do manual verification
var info = admin.getContractInfo(address) //lookup, fetch, decode
var source = info.source;
var abiDef = info.abiDefinition
```

这项工作的潜在机制具体如下：

- 能够公开访问的 URI 将合约信息上传到可辨认的地方。
- 任何人都可以只知道合约地址就能找到是什么 URI。

仅通过上述两个步骤的区块链注册就可以实现这些要求。第一步是在被称作 HashReg 的合约中用内容散表注册合约代码（散表）。第二步是在 UrlHint 合约中用内容散表注册一个 URL。这些注册合约是 Frontier 版本的一部分，已经参与到 Homestead 中。

要通过合约地址来查询 URL，获取实际合约元数据信息包，使用这一机制就足够了。

如果你是一个尽职的合约创建者，那么请遵循以下步骤。

1）将合约本身部署到区块链。

2）获取合约信息 JSON 文件。

3）将合约信息 JSON 文件部署到你选择的任意 URL。

4）注册代码散表→内容散表→ URL。

JS API 通过提供助手把这个过程变得非常容易。调用 admin.register 从合约中提取信息，在指定文件中写出 JSON 序列，运算文件的内容散表，最终将这个内容散表注册到合约代码散表。一旦将那个文件部署到任意 URL，你就能用 admin.registerUrl 来注册 URL 和你区块链上的内容散表（注意，一旦固定的内容选址模式被用作文件商店，url-hint 就不再有必要了）。

```
source = "contract test { function multiply(uint a) returns(uint d) { return a
*
```

```
    7; } }"
    // compile with solc
    contract = eth.compile.solidity(source).test
    // create contract object
    var MyContract = eth.contract(contract.info.abiDefinition)
    // extracts info from contract, save the json serialisation in the given file,
    contenthash = admin.saveInfo(contract.info, "~/dapps/shared/contracts/test/
info.json")// send off the contract to the blockchain
    MyContract.new({from: primaryAccount, data: contract.code}, function(error,
contract){
    if(!error && contract.address) {
    // calculates the content hash and registers it with the code hash in `HashReg`
    // it uses address to send the transaction.
    // returns the content hash that we use to register a url
    admin.register(primaryAccount, contract.address, contenthash)
    // here you deploy ~/dapps/shared/contracts/test/info.json to a url
    admin.registerUrl(primaryAccount, hash, url)
    }
    });
```

5.7.8　测试合约和交易

在为交易和合约排除故障时，你通常会需要一些低级的测试策略。本节将会介绍一些你可以用到的排错工作和做法。为了测试合约和交易而不产生实际的后果，最好在私有区块链上进行测试。这可以通过配置一个替代网络 ID（选择一个特别的数字）和 / 或不能用的端点来实现。推荐的做法是，为了测试，你用一个替代数据目录和端口，这样就不会意外地和实时运行的节点发生冲突（假定用默认运行。在虚拟机排错模式下开启 geth，推荐性能分析和最高的日志冗余级别）：

```
    geth --datadir ~/dapps/testing/00/ --port 30310 --rpcport 8110 --networkid
4567890 --nodiscover -
```

提交交易之前，你需要创建私有测试链（参阅测试网络相关章节），示例代码如下：

```
    // create account. will prompt for password
    personal.newAccount();
    // name your primary account, will often use it
    primary = eth.accounts[0];
    // check your balance (denominated in ether)
    balance = web3.fromWei(eth.getBalance(primary), "ether");
    // assume an existing unlocked primary account
    primary = eth.accounts[0];
    // mine 10 blocks to generate ether
    // starting miner
    miner.start(4);
    // sleep for 10 blocks (this can take quite some time).
    admin.sleepBlocks(10);
    // then stop mining (just not to burn heat in vain)
```

```
miner.stop();
balance = web3.fromWei(eth.getBalance(primary), "ether");
```

创建交易之后，你可以用下面的命令来强制运行：

```
miner.start(1);
admin.sleepBlocks(1);
miner.stop();
```

你也可以用以下的命令来查看即将发生的交易：

```
// shows transaction pool
txpool.status
// number of pending txs
eth.getBlockTransactionCount("pending");
// print all pending txs
eth.getBlock("pending", true).transactions
```

如果你提交合约创建交易，那么你可以检查想要的代码实际上是否嵌入到了当前的区块链：

```
txhash = eth.sendTansaction({from:primary, data: code})
// ... mining
contractaddress = eth.getTransactionReceipt(txhash);
eth.getCode(contractaddress)
```

5.8 如何部署、调用智能合约

5.8.1 RPC

5.7 节中已经讲到了如何写合约、部署合约及与合约互动。下面就来讲讲与以太坊网络和智能合约沟通的细节。

一个以太坊节点提供一个 RPC 界面。这个界面为 Đapp（去中心化应用）访问以太坊区块链的权限和节点提供功能，比如编译智能合约代码，它用 JSON-RPC 2.0 规范（不支持提醒和命名的参数）的子集作为序列化协议，在 HTTP 和 IPC（Linux/OS X 上的 Unix 域接口，在 Windows 上称为 pipe's）上可用。

5.8.2 惯例

RPC 界面会使用一些惯例，但它们不是 JSON-RPC 2.0 规范的一部分，这些惯例具体如下。

- 数字是十六进制编码。做这个决定是因为有些语言对运行极大的数字没有或只有很少的限制。为了防止这些错误数字类型是十六进制编码，由开发者来分析这些数字并正确处理它们。在维基页百科中查看十六进制编码章节可查看相关案例。

- 默认区块数字。几个 RPC 方法接受区块数字。在一些情况下，给出区块数字是不可能的，或者不太方便。在那样的情况下，默认区块数字可以是以下字符串中的一个["earliest"，"latest"，"pending"]。在维基百科中可查看使用默认区块参数的 RPC 方法列表。

5.8.3 部署合约

我们会通过不同的步骤来部署下面的合约，但只用到 RPC 界面：

```
contract Multiply7 {
event Print(uint);
function multiply(uint input) returns (uint) {
Print(input
*
7);
return input
*
7;
}
}
```

要做的第一件事就是确保 HTTP RPC 界面可用。这意味着我们在开始为 geth 提供 --rpc 标志、为 eth 提供 -j 标志。这个例子中使用的是私有开发链上的 geth 节点。通过这种方法，我们就不需要真实网络上的以太币了。

```
> geth --rpc --dev --mine --minerthreads 1 --unlock 0 console 2>>geth.log
```

这会在 http://localhost:8545 上启动 HTTP RPC 界面。

geth 支持 CORS 查看 --rpccorsdomain 标志以了解更多。

我们可以通过用 curl 检索 coinbase 地址和余额来证明界面正在运行。请注意这些例子中的数据在你本地的节点上会有所不同。如果你想要尝试这些参数，那么可视情况替换需要的参数：

```
> curl --data '{"jsonrpc":"2.0","method":"eth_coinbase", "id":1}'
localhost:8545
{"id":1,"jsonrpc":"2.0","result":["0xeb85a5557e5bdc18ee1934a89d8bb402398ee26a"]}
> curl --data '{"jsonrpc":"2.0","method":"eth_getBalance", "params": ["0xeb85a5
557e5bdc18ee1934a89d8bb402398ee26a"], "id":2}' localhost:8545
{"id":2,"jsonrpc":"2.0","result":"0x1639e49bba16280000"}
```

大家是否记得前面说过数字是十六进制编码？在这个情况下，余额作为十六进制字符串以 wei 的形式返还。如果希望以以太币为单位显示余额，那么可以从控制台使用 Web3，示例代码如下：

```
> web3.fromWei("0x1639e49bba16280000", "ether")
"410"
```

我们在私有开发链上有一些以太币，现在就可以部署合约了。第一步是验证 Solidity 编译器是否可用，可以用 eth_getCompilers RPC method 方法来检索可用的编译器，示例代码如下：

```
> curl --data '{"jsonrpc":"2.0","method": "eth_getCompilers", "id": 3}' local-
host:8545
{"id":3,"jsonrpc":"2.0","result":["Solidity"]}
```

我们可以看到 Solidity 编译器是可用的。

下一步就是把 Multiply7 合约编译到可以发送给以太坊虚拟机的字节代码中，示例代码如下：

```
> curl --data '{"jsonrpc":"2.0","method": "eth_compileSolidity", "params": ["contract
Multiply7 { event Print(uint); function multiply(uint input) returns (uint) {
Print(input
{"id":4,"jsonrpc":"2.0","result":{"Multiply7":{"code":"0x6060604052605f80601060
00396000f360606040
```

我们有了编译代码，现在就是需要决定花多少 gas 去部署它了。RPC 界面有 eth_estimateGas 方法，会给我们一个预估数量，示例代码如下：

```
> curl --data '{"jsonrpc":"2.0","method": "eth_estimateGas", "params": [{"from":
"0xeb85a5557e5bdc18ee1934a89d8bb402398ee26a", "data": "0x6060604052605f806010600039
6000f3606060405260e060020a6000350463c6888fa18114601a575b005b60586004356007810260609
081526000907f24abdb5865df5079dcc5ac590ff6f01d5c16edbc5fab4e195d9febd1114503da906020
90a15060070290565b5060206060f3"}], "id": 5}' localhost:8545
{"id":5,"jsonrpc":"2.0","result":"0xb8a9"}
```

最后部署合约：

```
> curl --data '{"jsonrpc":"2.0","method": "eth_sendTransaction", "params":
[{"from": "0xeb85a5557e5bdc18ee1934a89d8bb402398ee26a", "gas": "0xb8a9", "data":
"0x6060604052605f806010600396000f3606060405260e060020a6000350463c6888fa18114601
a575b005b60586004356007810260609081526000907f24abdb5865df5079dcc5ac590ff6f01d5c
16edbc5fab4e195d9febd1114503da90602090a15060070290565b5060206060f3"}], "id": 6}'
localhost:8545
{"id":6,"jsonrpc":"2.0","result":"0x3a90b5face52c4c5f30d507ccf51b0209ca628c6824
d0532bcd6283df7c08
```

节点将接受交易，并返还交易散表。我们可以用这个散表来跟踪交易。

下一步是决定部署合约的地址。每个执行的交易都会创建一个接收。这个接收包含交易的各种信息，比如交易被包含在哪个区块，以太坊虚拟机用掉了多少 gas 等。如果交易创建了一个合约，那么它也会包含合约的地址。我们可以用 eth_getTransactionReceipt RPC 方法检索接收，示例代码如下：

```
> curl --data '{"jsonrpc":"2.0","method": "eth_getTransactionReceipt", "params":
["0x3a90b5face52c4c5f30d507ccf51b0209ca628c6824d0532bcd6283df7c08a7c"], "id": 7}'
localhost:8545
{"id":7,"jsonrpc":"2.0","result":{"transactionHash":"0x3a90b5face52c4c5f30d507c
cf51b0209ca628c682
```

可以看到，在 0x6ff93b4b46b41c0c3c9baee01c255d3b4675963d 上创建了合约。如果你得到了零而不是接收，则说明还没有被纳入区块。这时，要检查看看你的矿工是否正在运行，然后再重新试一遍。

5.8.4　和智能合约互动

现在已经部署了合约，下面我们就可以和它互动了。有两种方法进行互动，即发送交易或像 5.7.6 节说明的那样调用。本节的例子中，将会发送交易到合约的 Multiply 方法里。

在我们的实例中，需要具体说明 from、to 和 data 参数。from 是我们账户的公共地址，to 是合约地址，data 参数有一点复杂，它包括了规定调用哪个方法和哪个参数的负载量。这就需要 ABI 发挥作用了，ABI 规定了如何为以太坊虚拟机规定和编码数据。

负载量的字节是功能选择符，其规定了调用哪个方法。它选取了 Keccak 散表的头 4 个字节，涵盖了功能名称参数类型，并进行十六进制编码。Multiply 功能接受一个参数，示例代码如下：

```
> web3.sha3("multiply(uint256)").substring(0, 8)
"c6888fa1"
```

下一步就是编码参数。我们只有一个 unit256，假定提供了值 6。ABI 有一个章节规定了编码 uint 字节的方法，如下：

```
int<M>: enc(X) is the big-endian two's complement encoding of X, padded on the
higher-oder (left) side with 0xff for negative X and with zero 字节 s for positive X
such that the length is a multiple of 32 bytes.
```

它会编码到 0000000000000000000000000000000000000000000000000000000000000006。

将功能选择符和编码参数结合起来，数据就会变成 0xc6888fa1000000000000000000000000000000000000000000000000000000000000006。

下面我们就来试一下：

```
> curl --data '{"jsonrpc":"2.0","method": "eth_sendTransaction", "params": [{"from":
"0xeb85a5557e5bdc18ee1934a89d8bb402398ee26a", "to": "0x6ff93b4b46b41c0c3c9baee01c255d3b467596
3d", "data": "0xc6888fa1000000000000000000000000000000000000000000000000000000000000006"}],
"id": 8}' localhost:8545
{"id":8,"jsonrpc":"2.0","result":"0x759cf065cbc22e9d779748dc53763854e5376eea074
09e590c990eafc0869
```

由于我们发送了交易，于是有交易散表返回。如果检索接收，则可以看到一些新内容，示例代码如下：

```
{
blockHash: "0xbf0a347307b8c63dd8c1d3d7cbdc0b463e6e7c9bf0a35be40393588242f01d55",
blockNumber: 268,
contractAddress: null,
cumulativeGasUsed: 22631,
gasUsed: 22631,
logs: [{
address: "0x6ff93b4b46b41c0c3c9baee01c255d3b4675963d",
blockHash: "0xbf0a347307b8c63dd8c1d3d7cbdc0b463e6e7c9bf0a35be40393588242f01d55",
blockNumber: 268,
data: "0x000000000000000000000000000000000000000000000000000000000000002a",
logIndex: 0,
topics: ["0x24abdb5865df5079dcc5ac590ff6f01d5c16edbc5fab4e195d9febd1114503da"],
transactionHash: "0x759cf065cbc22e9d779748dc53763854e5376eea07409e590c990eafc0869d74",
transactionIndex: 0
}],
transactionHash: "0x759cf065cbc22e9d779748dc53763854e5376eea07409e590c990eafc0869d74",
transactionIndex: 0
}
```

接收包含一个日志。日志由以太坊虚拟机在交易执行时生成，包含接收。如果我们查看 Multiply 功能，可以看到打印事件和输入次数 7 一起被提出。由于打印事件的参数是 uint256，因此可以根据 ABI 规则对它进行编码，这样就会得到预期的十进制 42。

```
> web3.sha3("Print(uint256)")
"24abdb5865df5079dcc5ac590ff6f01d5c16edbc5fab4e195d9febd1114503da"
```

这只是对一些最常见任务的简单介绍。可在 RPC 维基页面查看可用 RPC 方法的完整列表。

5.8.5 Web3.js

正如在之前的案例中所见的，使用 JSON-RPC 界面相当单调乏味且容易出错，尤其是在处理 ABI 的时候。Web3.js 是 JavaScript 库，它的目标是提供更友好的界面，减少出错的机会。

用 Web3 部署 Multiply7 合约看起来就像下面这样：

```
var source = 'contract Multiply7 { event Print(uint); function multiply(uint
input) returns (uint) { Print(input
var compiled = web3.eth.compile.solidity(source);
var code = compiled.Multiply7.code;
var abi = compiled.Multiply7.info.abiDefinition;
web3.eth.contract(abi).new({from: "0xeb85a5557e5bdc18ee1934a89d8bb402398ee26a",
data: code}, function (err, contract) {
if (!err && contract.address)
console.log("deployed on:", contract.address);
}
);
```

```
deployed on: 0x0ab60714033847ad7f0677cc7514db48313976e2
```

装载一个部署的合约，发送交易，示例代码如下：

```
var source = 'contract Multiply7 { event Print(uint); function multiply(uint
input) returns (uint) { Print(input
var compiled = web3.eth.compile.solidity(source);
var Multiply7 = web3.eth.contract(compiled.Multiply7.info.abiDefinition);
var multi = Multiply7.at("0x0ab60714033847ad7f0677cc7514db48313976e2")
multi.multiply.sendTransaction(6, {from: "0xeb85a5557e5bdc18ee1934a89d8bb402398
ee26a"})
```

注册一个回调，打印事件创建日志的时候会调用：

```
multi.Print(function(err, data) { console.log(JSON.stringify(data)) })
{"address":"0x0ab60714033847ad7f0677cc7514db48313976e2","args": {"":"21"},"bloc
kHash":"0x259c7dc0
```

在 Web3.js 维基页面中可查看更多信息。

5.8.6　控制台

geth 控制台提供命令行界面和 Javascript 执行时间。它可以连接到本地或远程的 geth 或 eth 节点。它会装载用户能够使用的 Web3.js 库，从而方便用户从控制台通过 Web3.js 部署智能合约，并与智能合约互动。实际上 Web3.js 章节（5.8.5 节）的例子可以被复制进控制台并且调用。

5.8.7　查看合约与交易

有几个可用的在线区块链浏览器，能让你查询以太坊区块链，它们分别是：

- EtherChain
- EtherCamp
- EtherScan

其他可查看节点或交易的资源：

- EtherNode：节点的地理分配，由客户端区分。
- EtherListen：实时以太坊交易可视器和可听器。

5.9　智能合约案例实战

以太坊是区块链开发领域最好的编程平台，而 truffle 是以太坊（Ethereum）最受欢迎的一个开发框架，这也是本节介绍 truffle 的原因。实战是最重要的事情，本章不讲原理，只搭建环境，运行第一个区块链程序（Dapp）。

1. 安装 truffle

安装 truffle 的命令如下：

```
$ npm install -g truffle
```

2. 依赖环境

可用的系统包括：Windows、Linux 和 Mac OS X，推荐使用 Mac OS X，不建议使用 Windows，因为会碰到各种各样的问题，很可能会导致放弃。首先，访问 https://nodejs.org 官方网站下载安装 NodeJS。

此外，需要安装 Ethereum 客户端，来支持 JSON RPC API 调用。

至于开发环境，推荐使用 EthereumJS TestRPC，地址为 https://github.com/ethereumjs/testrpc。

安装命令如下：

```
$ npm install -g ethereumjs-testrpc
```

3. 新建第一个项目

通过以下命令新建一个项目：

```
$ mkdir zhaoxi
$ cd zhaoxi
$ truffle init
```

默认会生成一个 MetaCoin 的 demo，可以从这个 demo 中学习 truffle 的架构。

项目的目录结构如图 5-3 所示。

项目所有文件的目录如图 5-4 所示。

```
/Users/bob/zhaoxi/
▸ app/
▸ contracts/
▸ environments/
▸ test/
  truffle.js
```

图 5-3 项目的目录结构

```
" Press ? for help

.. (up a dir)
/Users/bob/zhaoxi/
▾ app/
  ▾ images/
  ▾ javascripts/
      app.js
  ▾ stylesheets/
      app.css
    index.html
▾ contracts/
    ConvertLib.sol
    MetaCoin.sol
▾ environments/
  ▾ development/
      config.js
  ▾ production/
      config.js
  ▾ staging/
      config.js
  ▾ test/
      config.js
▾ test/
    metacoin.js
  truffle.js
```

```
1  import "ConvertLib.sol";
2
3  // This is just a simple example of a coin-like contract.
4  // It is not standards compatible and cannot be expected to talk to other
5  // coin/token contracts. If you want to create a standards-compliant
6  // token, see: https://github.com/ConsenSys/Tokens. Cheers!
7
8  contract MetaCoin {
9    mapping (address => uint) balances;
10
11   function MetaCoin() {
12     balances[tx.origin] = 10000;
13   }
14
15   function sendCoin(address receiver, uint amount) returns(bool sufficient) {
16     if (balances[msg.sender] < amount) return false;
17     balances[msg.sender] -= amount;
18     balances[receiver] += amount;
19     return true;
20   }
21   function getBalanceInEth(address addr) returns(uint){
22     return ConvertLib.convert(getBalance(addr),2);
23   }
24     function getBalance(address addr) returns(uint) {
25       return balances[addr];
26     }
27 }
```

图 5-4 项目文件目录结构

现在，通过以下命令编译项目：

```
$ truffle compile
```

图 5-5 是运行以上命令后的结果。

```
bob@192 zhaoxi % truffle compile
Checking sources...
Compiling ConvertLib.sol...
Compiling MetaCoin.sol...
Writing contracts to ./environments/development/contracts
```

图 5-5　truffle compile 执行结果图

下面介绍部署项目的方式。

部署之前先启动 TestRPC，命令如下：

```
$ testrpc
$ truffle deploy（在 truffle 2.0 以上版本中，命令变成了：truffle migrate）
```

图 5-6 是运行 truffle deploy 后的结果。

```
bob@192 zhaoxi % truffle deploy
Using environment development.
No contracts updated; skipping compilation.
Collecting dependencies...
Deployed: ConvertLib to address: 0x1b04bf4deeb1b0b7ba8e8e50f2bc157708f2e20b
Linking Library: ConvertLib to contract: MetaCoin at address: 0x1b04bf4deeb1b0b7ba8e8e50f2bc157708f2e20b
Deployed: MetaCoin to address: 0x0baeb7b99fd63f197865cb7b4d66406bd7182a4d
Writing built contract files to ./environments/development/contracts
```

图 5-6　truffle deploy 执行结果图

$ truffle migrate migrate 的执行结果见图 5-7。

```
bob@192 zhaoxi % truffle migrate
Running migration: 1_initial_migration.js
  Deploying Migrations...
  Migrations: 0x34e94dd89ed596077d5a4f9d8481dbd0633a454c
Saving successful migration to network...
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Deploying ConvertLib...
  ConvertLib: 0x62038717aeb35113425683d8eb3dfab65239162c
  Linking ConvertLib to MetaCoin
  Deploying MetaCoin...
  MetaCoin: 0xbeb2aaac8e7c85426dcb120667d5fe49358581b0
Saving successful migration to network...
Saving artifacts...
```

图 5-7　truffle migrate migrate 执行结果图

现在，可以启动服务了，命令如下：

```
$ truffle serve
```

图 5-8 是 truffle serve 执行结果图。

```
bob@192 zhaoxi % truffle serve
Using environment development.
Serving app on port 8080...
Rebuilding...
Completed without errors on Sun May 01 2016 05:53:12 GMT+0800 (CST)
```

图 5-8　truffle serve 执行结果图

启动服务后，就可以在浏览器访问该项目了，地址是：http://localhost:8080/，网页界面

如图 5-9 所示。

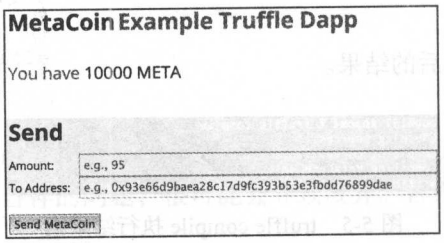
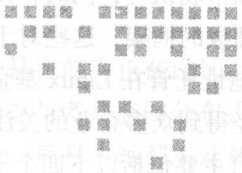


图 5-9 智能合约运行界面

好了，第一个区块链程序运行起来了，后面就可以不断地实践深入学习了。

参考资料

[1] https://en.wikipedia.org/wiki/Hashcash
[2] https://en.bitcoin.it/wiki/Proof_of_work
[3] https://en.bitcoin.it/wiki/Block_hashing_algorithm
[4] https://en.bitcoin.it/wiki/Difficulty
[5] https://en.bitcoin.it/wiki/Target
[6] https://en.bitcoin.it/wiki/Address
[7] https://en.bitcoin.it/wiki/Transaction
[8] https://en.bitcoin.it/wiki/Script
[9] https://en.bitcoin.it/wiki/Network
[10] https://en.bitcoin.it/wiki/Block
[11] https://en.bitcoin.it/wiki/API_reference_(JSON-RPC)
[12] https://github.com/bitcoin/bitcoin.git
[13] http://yinyanghu.github.io/posts/2013-07-21-bitcoin.html
[14] https://www.zhihu.com/question/20941124/answer/16668373
[15] https://biqu.io/t/topic/59
[16] https://biqu.io/t/topic/58
[17] https://biqu.io/t/topic/57
[18] https://biqu.io/t/topic/55
[19] https://biqu.io/t/topic/56
[20] https://en.bitcoin.it/wiki/Technical_background_of_version_1_Bitcoin_addresses
[21] http://www.righto.com/2014/02/bitcoins-hard-way-using-raw-bitcoin.html
[22] http://www.righto.com/2014/02/bitcoin-mining-hard-way-algorithms.html



第 6 章 Chapter 6

Fabric 原理和实操

超级账本（Hyperledger）项目是致力于推进区块链数字技术和交易验证的开源项目，目标是让开源社区成员共同合作，共建开放平台，从而满足来自多个不同行业的用户需求，并简化业务流程。通过创建分布式账本的公开标准，实现虚拟和数字形式的价值交换。

6.1 超级账本项目背景

区块链已经成为当下最受人关注的开源技术，然而当前的区块链技术还比较难以理解和实现，而且缺乏统一的规范和标准。

2015 年 12 月，Linux 基金会牵头联合 30 家初始成员（包括 IBM、Accenture、Intel、J.P.Morgan、R3、DAH、DTCC、FUJITSU、HITACHI、SWIFT、Cisco 等）共同宣告了 Hyperledger 项目的成立。该项目试图打造一个透明、公开、去中心化的超级账本项目，作为区块链技术的开源规范和标准，让更多的应用能更容易地建立在区块链技术之上。目前已经有超过 80 家企业和机构（大部分均为各自行业的领导者）宣布加入 Hyperledger 项目，其中，来自我国的区块链技术公司超过了 20 家。

如果说以比特币为代表的货币区块链技术为 1.0，以以太坊为代表的合约区块链技术为 2.0，那么实现了完备的权限控制和安全保障的 Hyperledger 项目毫无疑问代表着区块链技术 3.0 时代的到来。

IBM 贡献了数万行已有的 Open Block Chain 代码，Digital Asset 则贡献了企业和开发者的相关资源，R3 贡献了新的金融交易架构，Intel 也贡献了分布式账本相关的代码。

该项目的出现，实际上是宣布区块链技术已经不再是一个单纯的开源技术了，它已经正

式被主流机构和市场认可；同时，Hyperledger 首次提出并实现了完备的权限管理、创新的一致性算法和可插拔的框架，这些对于区块链相关技术和产业的发展都将产生深远的影响。

项目官方地址托管在 Linux 基金会网站，代码托管在 Gerrit 上，并通过 Github 提供代码镜像，目前已经得到众多企业的关注。

目前该项目主要包括以下四个子项目。

- Fabric：包括 Fabric 和 Fabric-api、Fabric-sdk-node、Fabric-sdk-JAVA、Fabric-sdk-py、Fabric-chaintool 等，目标是区块链的基础核心平台，支持 Kafka 和 PBFT 等模块化共识算法，支持权限管理和隐私保护（如图 6-1 所示），最早由 IBM、DAH 及 Blockstream 发起。
- Sawtooth Lake：是 Intel 主要贡献和主导的区块链平台，包括 arcade、core、dev-tools、validator、mktplace 等，支持全新的基于硬件芯片的共识机制 Proof of Elapsed Time（PoET）。
- Iroha：分布式账本平台项目，主要由 Soramitsu 发起和贡献，特点是支持模块化和移动应用。
- Blockchain-explorer：由 DTCC 牵头发起，提供 Web 操作界面，通过界面可快速查看、查询超级账本区块链的信息、状态等。

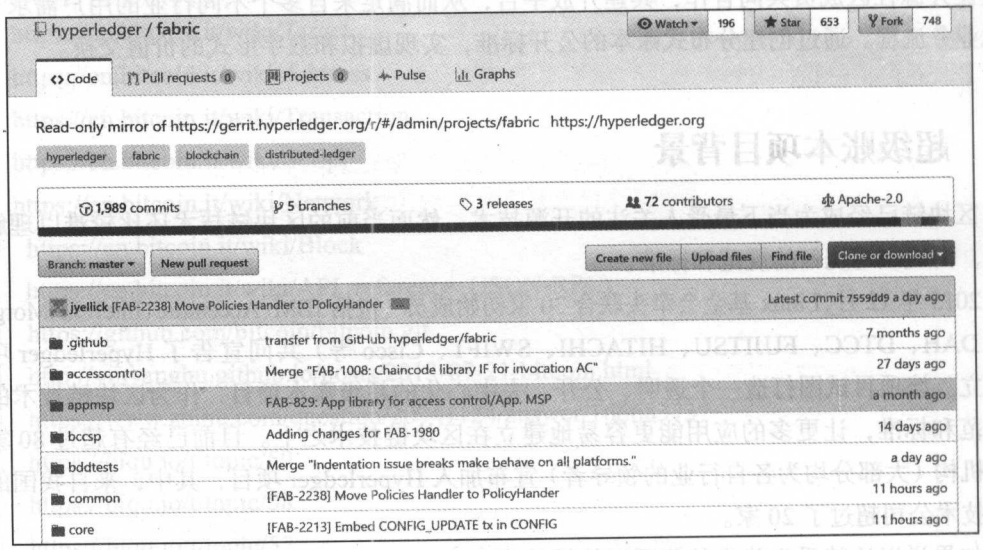


图 6-1 Hyperledger Fabric 项目

目前，所有项目均处于孵化状态，项目约定共同遵守的基本原则具体如下。

- 重视模块化设计，包括交易、合同、一致性、身份、存储等技术场景。
- 代码可读性，保障新功能和模块都可以很容易添加和扩展。
- 发展路线，随着商业化需求的深入和应用场景的丰富，不断增加和演化新的项目。

6.2　Fabric 简介

在上面介绍的 4 个子项目中，我们最关注 Fabric，它是目前为止在设计上最贴近联盟链思想的区块链。Fabric 是基于数字事件、交易调用、不同参与者共享的分布式总账技术。分布式总账只能通过共识的参与者来更新，而且一旦被记录，信息永远都不能被修改。记录的每一个事件都可以根据参与者的协议进行加密验证。

交易是安全、私有并且可信的。每个参与者都需要通过向成员权限管理服务证明自己的身份来访问系统。交易是通过发放在网络上完全匿名的证书来生成的。交易内容通过复杂的密钥加密来保证只有参与者才能看到，以确保业务的私密性。

分布式总账可以按照相关规则来接受全部或部分审计。在与参与者的合作中，审计员可以通过基于时间的证书来获得总账的查看权限，并且可通过连接交易来获取实际交易方之间的资产操作。

Fabric 是区块链技术的一种实现，它通过模块化的架构允许组件进行“插入 - 运行”来实现这份协议规范。它具有强大的容器技术，因此可支持任何主流的语言来开发智能合约。

Fabric 是基于工业化、商业化的需求来设计的，并引入了可扩展性。它是许可性区块链技术方案，隐私保护、数据保密是整个区块链网络的核心要素。

本节将描述区块链技术框架 Fabric 1.0 的基础架构设计，Fabric 的基础架构中会涉及不同角色的区块链节点，主要包括维护状态和总账的节点（peer 节点）、审批交易顺序的共识节点（orderer 节点），审批后的这些交易包含在总账中。在传统的区块链架构中（包括 0.6 版本和更早版本的 Fabric 框架实现），这些角色都是统一到验证节点的实现（参考 Hyperledger Fabric 0.6 版本的验证节点）。新版本的架构引入了背书节点（背书者），作为一种特殊类型的区块链节点，其负责模拟交易的执行和背书处理（大体上相当于 Fabric 0.6 版本的执行交易）。

与 Hyperledger Fabric 0.6 版本的节点 / 共识机制 / 背书者的统一设计相比，新版本的架构还具有如下优点。

- 链代码信任机制灵活：新架构将排序的信任假定和链代码（也就是区块链应用）的信任假定独立开来。换句话说，就是一堆节点（排序节点）能够提供排序服务（ordering service）并且允许部分节点失败或行为异常，针对每个链代码，背书者均可以是不同的。
- 扩展性：因为负责特定链代码的背书节点（endorser）对于排序者是正交的，相比在 Hyperledger Fabric 0.6 版本中，这些功能集中在同一个 VP 节点（验证节点），新版本的架构扩展性更好。尤其是当不同的链代码（chaincode）指定不同的背书者（endorser）时，这些背书者引入了在不同背书者之间的链代码分区，并且允许并行的链代码执行（背书处理）。此外，由于链代码执行的代价可能会很高，因此将它从排序服务的关键路径中移除，也就是说，在新的架构中，链代码的执行和共识处理分开了，这样做的好处是可以独立扩展，链代码执行成本更高，也不会影响共识处理。

- 机密性：新的架构使得部署机密性需求的链代码变得更加容易。这里的机密性需求主要是指交易的内容和状态更新有机密性需求。
- 共识模块化：新的架构针对共识机制的设计更加灵活，允许可插入式的共识实现，如排序服务等。

6.3 系统架构

Fabric 区块链是一种由多个相互通信的节点组成的分布式系统，其上运行着一种称为链代码（chaincode），或者称为智能合约（Smart contract）的程序，这段程序的主要功能是保存状态和账本数据，执行交易。链代码是主要的研究对象，因为交易是在链代码上被调用的业务操作的。交易必须进行背书处理，而且只有背书过的交易才能被提交并对状态产生影响。Fabric1.0 架构中存在一个或多个特殊的链代码，这些链代码主要用于管理功能，总体上被称为系统链代码。

6.3.1 交易

交易可能有如下两种类型。

- 部署交易（deploy transaction）：部署交易创建新的链代码，并且用一个程序作为参数。当一个部署交易成功执行时，链代码被安装到区块链上。
- 调用交易（invoke transaction）：调用交易在先前部署的交易上下文中执行操作。调用交易指的是链代码和它提供的一个或多个功能。当调用交易成功地执行时，链代码执行指定的函数，这些函数执行时可能会修改相应的状态，并返回输出。

正如接下来将要描述的，部署交易是调用交易的特殊情况。部署交易创建了新的链代码，其相当于是系统链代码上的一个调用交易。

说明 本章当前假定一个交易要么创建新的链代码（deploy transaction），要么调用一个已经部署的链代码提供的操作（invoke transaction）。以下内容不在本章的介绍范围之内。

- 查询交易优化（1.0 版本包含）
- 支持跨链代码的交易（1.0 后续版本的特征）

6.3.2 区块链数据结构

1. 状态

区块链的最新状态被模型化为一个 key/value 数据库存储。key 是名字，value 是任意的 blob 类型的值。运行在区块链上的 chaincode 应用程序通过调用 chaincode 接口的 put/get 方法来对这些状态进行操作。状态被持久化存储，会将状态的更新记录成日志。注意，版本化

的 KVS 是作为状态模型采用的，其实现可以是实际 key/value 存储，也可以是关系型数据库或其他解决方案。

正式地说，状态被模型化为一个 K –> (V X N) 映射的元素，其中：

- K 是键的集合。
- V 是值的集合。
- N 是无数个有序版本号的集合。单射函数 next: N –> N 取一个 N 的元素，返回下一个版本号。

V 和 N 都有一个特殊的元素 \bot，代表 N 值最小的元素。初始化的时候，所有的键都被映射到（\bot,\bot）。s(k)=(v,ver) 这个表达式中，v 用 s(k).value 来表示，ver 用 s(k).version 来表示。

KVS 的操作其定义如下。

- put(k,v) 操作：对 k\in K 和 v\in V 键值对，区块链状态 s 的新状态 s' 计算方法是：s'(k)=(v,next(s(k).version))。并且对所有的 k'!=k，表达式 s'(k')=s(k') 都成立。
- get(k) 操作：返回 s(k)。

状态被节点（peer）维护，而不是被排序者（orderer）和客户端维护。

状态分区。KVS 中的键可以通过名称识别出它们属于哪个链码（chaincode），所以只有特定链码的交易才能修改属于这个链码的键。原则上，任意的链码都能读取属于其他链码的键。V1.0 的后续版本会支持修改两个或多个链码状态的跨链交易。

2. 总账

总账（ledger）提供了所有成功的状态变化（合法交易）可核实的历史，也记录了不成功的状态变化尝试（非法交易）。这一切都发生在系统的运行期间。

总账可通过排序服务（ordering service）来创建，作为一种完全排序的 Hash 链，Hash 链由合法和非法交易的区块组成。Hash 链对总账中的区块进行了完全排序，每个区块包含一组完全排序的交易。这使得所有的交易都是完全排序的。

总账被保持在所有的节点中，同时视情况也可以保存在排序节点的子集中。在排序者的上下文中，总账指的是排序者总账（order ledger），而在节点上下文中，则指的是原始总账（peer ledger）。原始总账与排序者总账的区别在于，原始总账本地维护了一个位掩码可以用于区分合法交易和非法交易。

节点可以精简原始总账。排序者在容错性和原始总账的可用性方面维护着排序者总账，倘若维护的是排序服务的属性，那么它可以在任何需要的时候决定是否精简。总账（ledger）允许节点重播所有交易的历史，也允许节点重新构建状态。

6.3.3 节点

节点（peer）是区块链的通信实体。节点仅仅是一个逻辑概念，不同类型的多个节点可

以运行在同一个物理服务器上。节点的类型一共包括如下三种。

- 客户端或提交客户端（client or submitting-client）：客户端向背书者（endorser）提交一个实际的交易调用，并且向排序服务广播交易提案（trans-proposal）。
- 节点：节点提交交易，维护状态和总账（ledger）的副本。此外，节点还有一个特殊的背书者角色。
- 共识服务节点或投票者节点：一个运行通信服务的节点，实现了投递保证，如原子的和完全排序的广播。

那么，节点是怎么被分组成“信任域（trust domains）”的，又是如何与控制它们的逻辑实体进行关联的呢？下面就来详细解释各种节点类型。

1. 客户端

客户端（client）表示终端用户的实体，用于创建和调用交易，它能同时与节点及排序服务通信，不过，要实现通信必须先连接到一个与区块链通信的节点。事实上，客户端是可以连接到它可以选择的任何节点的。

2. 节点

节点接收来自排序服务（ordering service）的排序状态更新，状态更新以区块的形式进行存储，同时节点还用于维护状态和总账。

此外，节点还能够承担一个背书节点的特殊角色，或者称为背书者（endorser）。从功能上来说，背书节点主要用于响应特定的链代码的背书请求，在交易提交前对交易进行背书处理。每个链代码都可能指定一个背书策略，这些背书策略会涉及一个背书节点集合，这些策略定义了合法的交易背书的必要条件（典型的策略是一个背书者集合的签名）。

3. 排序服务节点（排序者）

排序者形成了排序服务，一个通信 Fabric 提供了投递保证。排序服务可以被实现为多种不同的方式：从一个中心化的服务（被用于开发和测试）到分布式的协议（目标定位在不同的网络和节点故障模型上）。

排序服务提供了通向客户端（client）和节点（peer）的共享的通信通道，提供了包含交易的消息广播服务（broadcast 和 deliver）。客户端连接到通道时，通过这个通道可以向所有的节点广播（broadcast）消息，通道可以向所有的区块链节点投递（deliver）消息。通道示意图如图 6-2 所示。

通道支持所有消息的原子投递，也就是说，消息通信全是排序投递，且会保证可靠性（特定于实现）。换句话说，通道会向所有连接到的节点输出同样的消息，并且以同样的逻辑顺序输出到所有节点。这种原子通信保证也称为全排序广播、原子广播或分布式系统上下文的共识。通信过的消息则会作为包含在区块链状态中的候选交易。

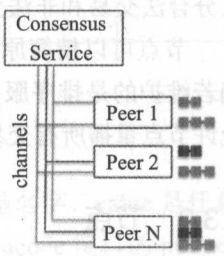


图 6-2 通道示意图

（1）通道分区（排序服务通道）

排序服务可以支持多通道，类似于发布 / 订阅消息系统中的主题（topic）。客户端能够连接到一个给定的通道，并通过给定的通道发送消息和接收到达的消息。通道可能会有分区，客户端连接到一个通道时是不知道其他通道存在的，但是客户端可以连接到多个通道。即使 Fabric 1.0 版本的一些排序服务实现了支持多个通道，但是为了简单表示，在本文的后续部分，仍然假定排序服务是由单个通道 / 主题组成的。

（2）排序服务 API

节点通过排序服务提供的接口连接到由共识服务提供的通道中。排序服务 API 由如下两个基本的操作组成（更通用的说法是异步事件）。

- 广播（broadcast）：客户端调用这个 API 在通道上广播任意的消息 blob。在 BFT 上下文中，当发送一个请求给服务时，也称为 request（blob）。
- 投递（deliver）：排序服务在节点上调用这个 API（seqno, prevhash, blob）投递消息，投递消息带有非负序列号 seqno 和最近一次发送消息的 Hash（prevhash）。换句话说，它是一个来自排序服务的输出事件。deliver() 在发布订阅系统中有时也称为 notify，在 BFT 系统中称为 commit。

（3）总账和区块构成

总账（参看 6.3.2 节下总账的相关内容）包含所有的排序服务的输出数据。它是 deliever 投递事件的序列，这些事件序列形成了 Hash 链。

（4）排序服务内容

一个广播了的消息发生了什么，投递了的消息之间存在什么关系？排序服务（或原子广播通道）包括如下的保证。

- 安全性（一致性保证）：只要节点能够足够长时间地连接到通道（它们可以断开或宕机，但是将会重启和重连），它们就能看到具有相同序号的投递消息（比如 seqno、prevhash、blob 等）。这意味着在所有的节点上，输出（也就是 deliever 事件）将以同样的顺序发生。相同的序号有着相同的内容，也就是 blob 和 prevhash 相同。注意，这仅仅是一个逻辑顺序，一个节点的 deliver(seqno, prevhash, blob) 并不需要和另外一个节点上输出了相同消息的 deliver(seqno, prevhash, blob) 有时间上的关联。换句话说，给定一个特定的 seqno，不会有两个正常的节点发送不同的 prevhash 和 blob。此外，除非某个客户端（某个节点）实际调用了 broadcast(blob)，否则不会投递 blob 值。而且，每个广播的 blob 只会投递一次。deliver 事件包含上一个 deliver 事件中的数据的加密 Hash。当排序服务实现原子广播保证时，prevhash 是序号为 seqno-1 的 deliver() 事件的加密 Hash。这就在不同的 deliver() 事件之间建立了一个 Hash 链，用来帮助验证共识输出的完整性。第一个 deliver 事件是一个特殊情况，prevhash 有一个默认值。
- 活跃度（投递保证）：排序服务的活跃度保证是排序服务实现指定的。精确的保证取决

于网络和节点的故障模型。

原则上，如果提交客户端没有失败，那么排序服务应该保证每个连接到排序服务的正常节点最终可以投递每个提交的交易。

- 总结：排序服务做了如下的保证。
- 协议（Agreement）：在正常节点上的任意两个事件（比如，deliver(seqno, prevhash0, blob0) 和 deliver(seqno, prevhash1, blob1)），如果有相同的序列号 seqno，则 prevhash0==prevhash1，同时 blob0==blob1。
- Hash 链完整性（Hash chain integrity）：在正常节点上的任意两个事件，deliver(seqno-1, prevhash0, blob0) 和 deliver(seqno, prevhash, blob)，有 prevhash = Hash (seqno1||prevhash0||blob0)。
- 不跳跃（No skipping）：如果排序服务在一个正常的节点上输出 deliver(seqno, prevhash, blob)，加入 seqno>0，那么意味着节点已经投递了一个事件 deliver(seqno-1, prevhash0, blob0)。
- 不创建（No creation）：在一个正常节点上的任意事件 deliver(seqno, prevhash, blob)，前面一定有一个节点发送了 broadcast(blob) 事件。
- 不重复（No duplication）：对于任意两个事件：broadcast(blob) 和 broadcast(blob')，当两个事件 deliver(seqno0, prevhash0, blob) 和 deliver(seqno1 prevhash1, blob') 发生在正常节点，且 blob == blob'，那么 seqno0==seqno1，且 prevhash0==prevhash。
- 活跃度（Liveness）：如果一个正常的客户端调用一个 broadcast(blob) 事件，然后每个正常的节点最终都会发布一个 deliver(*, *, blob) 事件，这里的 * 表示一个任意值。

4. Fabric 多通道技术

Hyperledger Fabric 1.0 架构会将共识服务和交易服务（总账）分开维护。Fabric 共识服务采用基于 topic 模型的发布订阅机制的多通道技术发布消息。通道是一个逻辑概念，通道相当于 Kafka 中的 topic 分区。共识服务是通过一堆称为 orderer 的实体来执行的，而总账是由 peer 来维护和管理的。

每个节点通过一个或多个通道连接到共识节点，就像发布订阅通信系统的客户端一样（如图 6-3 所示）。在一个通道上，广播消息通过共识排序，这样在同一个通道的所有的订阅者（节点）都以同样的顺序接收到同样的消息。交易以加密链接区块的方式分发到所有订阅的节点上。每个节点都会验证区块，提交区块到总账并为应用提供其他服务，应用可以通过这些服务使用总账。共识服务在不同通道中处理消息是独立的，跨通道的消息不能得到保证。

在图 6-3 中，peer 1、peer 2、peer 3、peer 5、peer 6 与通道 1 共同维护红色账本；peer1、peer 7、peer 8 与通道 2 共同维护绿色账本；peer 1、peer 2、peer 3 与通道 3 共同维护蓝色账本。这三套账本组成了三条不同的区块链，而这三个区块链又是相互独立的。

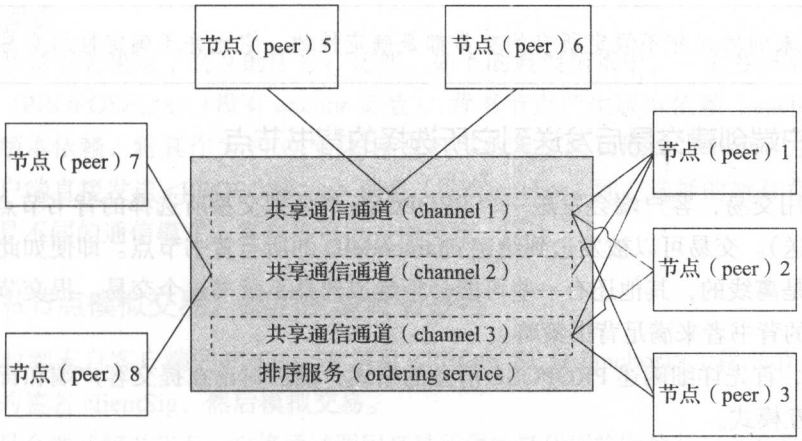


图 6-3　多通道技术示意图

6.4　交易背书的基本流程

接下来将概述一个交易的高层次请求流程。图 6-4 展示了背书交易的整体流程。

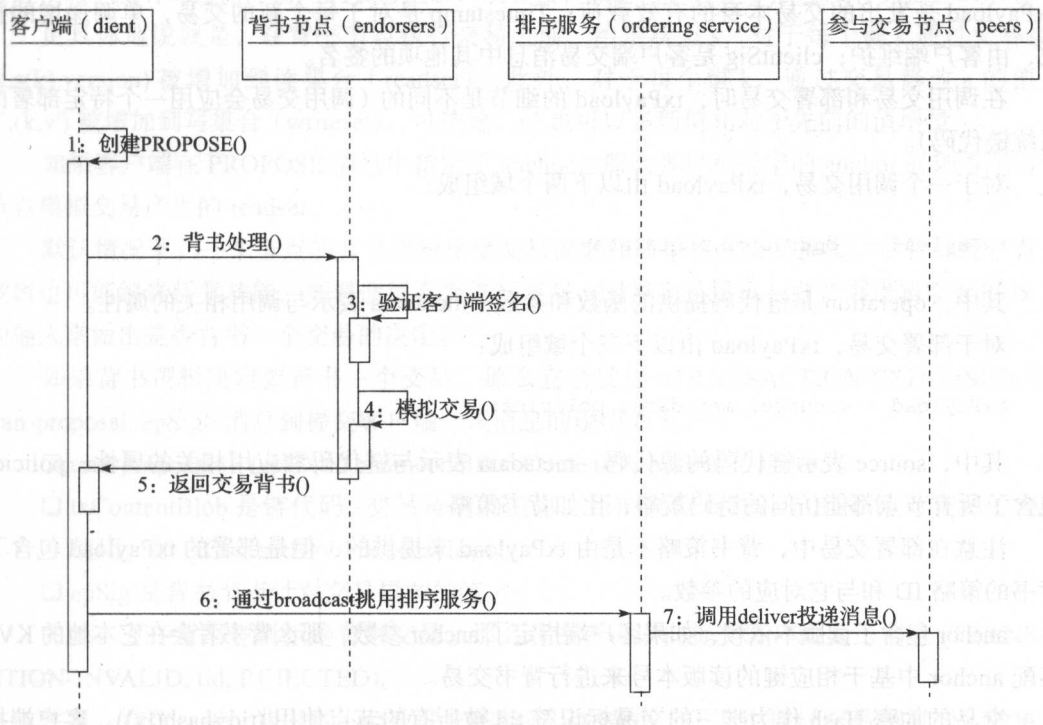


图 6-4　背书交易的整体流程

接下来的协议并不假定所有的交易都是确定性的，它允许不确定性的交易。

6.4.1 客户端创建交易后发送到它所选择的背书节点

为了调用交易，客户端会发送一个 PROPOSE 消息到交易所选择的背书节点集合（可能不是同时发送）。交易可以被发送到给定 chaincodeID 的所有背书节点。即便如此，一些背书者也可能会是离线的，其他还有一些可能会拒绝或选择不背书这个交易。提交节点应尽可能地利用可用的背书者来满足背书策略。

接下来，首先详细阐述 PROPOSE 消息的格式，然后讨论在提交客户端和背书者之间可能存在的交互模式。

1. PROPOSE 消息格式

PROPOSE 消息的格式是 <PROPOSE,tx,[anchor]>，其中 tx 是必须提供的，而 anchor 是可选的参数，下面会详细阐述。

```
tx=<clientID,chaincodeID,txPayload,timestamp,clientSig>
```

这里的 clientID 是提交客户端的 ID；chaincodeID 指的是交易所属的链代码的 ID；TxPayload 是发出的交易本身的有效载荷；Timestamp 是对于每个新的交易，单调递增的整数，由客户端维护；clientSig 是客户端交易消息中其他项的签名。

在调用交易和部署交易时，txPayload 的细节是不同的（调用交易会应用一个特定部署的系统链代码）。

对于一个调用交易，txPayload 由以下两个域组成：

```
txPayload = <operation, metadata>
```

其中，operation 是链代码提供的函数和参数；metadata 表示与调用相关的属性。

对于部署交易，txPayload 由以下三个域组成：

```
txPayload = <source, metadata, policies>
```

其中，source 表示链代码的源代码；metadata 表示与链代码和应用相关的属性；policies 包含了所有节点都能访问的链码策略，比如背书策略。

注意在部署交易中，背书策略不是由 txPayload 来提供的，但是部署的 txPayload 包含了背书的策略 ID 和与它对应的参数。

anchor 包含了读版本依赖，如果客户端指定了 anchor 参数，那么背书者会在它本地的 KVS 匹配 anchor 中基于相应键的读版本号来进行背书交易。

交易的加密 Hash 作为唯一的交易标识符 tid 被所有的节点使用 (tid=hash(tx))。客户端把 tid 存储在内存中，并且等待来自背书节点的响应。

2. 消息模式

客户端和背书者决定了交互的序号。比如，如下的典型场景中，一个客户端向单个的背书节点发送 <PROPOSE, tx>（没有 anchor 参数），背书节点产生版本依赖（anchor），客户端稍后会用到版本依赖，将其作为 PROPOSE 消息的参数发送给其他的背书者。另外再列举一个例子，客户端直接发送 <PROPOSE, tx> 消息（没有 anchor）到它选择的所有背书者上。有可能采用的是不同的通信模式，客户端可以自由决定。

6.4.2　背书节点模拟交易，然后生成背书签名

一旦接收到来自客户端的 PROPOSE 消息 <PROPOSE,tx,[anchor]>，背书节点 epID 首先验证客户端的签名 clientSig，然后模拟交易。

模拟交易会涉及背书节点，它将通过调用交易所属的链代码临时执行一个交易（txPayload），并执行背书节点本地持有的状态副本。

作为执行的结果，背书节点会计算读版本依赖（readset）和状态更新（writeset），在 DB 语言中也称之为 MVCC+postimage。

状态由键值对组成，所有的键值对条目都是版本化的，意思是说每个条目都包含排序后的版本信息，当存储在一个键下的值被更新时，这个排序的版本号每次都会增加。节点解释了通过链代码下所有的键值对访问的交易记录，或者读，或者写，但是节点还没有更新它的状态。

更具体地说就是，在背书节点执行交易之前，给定状态 s，对于每个键 k 通过交易读，(k,s(k).version) 被增加到读集合（readset）。此外，对于每个键 k，通过交易修改 k 的值为 v',(k,v') 被增加到写集合（writeset）。可选地，v' 也可以是新值相对于先前的值增量。

如果客户端在 PROPOSE 消息中指定了 anchor，那么客户端指定的 anchor 必须等于背书节点模拟交易产生的 readset。

默认情况下，一个节点的背书逻辑接受交易提案和简单签名交易提案。尽管如此，背书逻辑也可能解释任意功能，如遇遗留系统进行交互，附带交易提案，则背书逻辑需要以 tx 作为输入来做出是否背书一个交易的决定。

如果背书逻辑决定要背书一个交易，那么它会发送 <TRANSACTION-ENDORSED, tid, tran-proposal, epSig> 消息到提交客户端。该消息的说明如下。

- tran-proposal := (epID,tid,chaincodeID,txContentBlob,readset,writeset)
- txContentBlob 是链代码 / 交易特定的信息。目的是使 txContentBlob 作为交易的表示使用。如 txContentBlob=tx.txPayload。
- epSig 是背书节点针对交易提案的签名。

另外，万一背书逻辑拒绝背书交易，那么背书者可以向提交客户端发送消息 (TRANSACTION-INVALID, tid, REJECTED)。

注意，在这一步，背书者不会改变它的状态，在背书上下文中通过交易模拟产生的更新也不会影响状态。

6.4.3 提交客户端获取交易的背书，通过排序服务广播

之前，提交节点一直在等待，直到收到了“足够”的关于（TRANSACTION-ENDORSED, tid, *, *）的消息和签名，一旦收到这些信息，就可以得出交易已被背书的结论。这可能会涉及与背书节点之间一到多轮的交互。

“足够”的精确数字取决于链代码的背书策略。如果满足了背书策略，那么交易就算背书成功了。注意，这时候还没有提交交易。

如果提交客户端没有收集到交易提案的背书，那么它会放弃这次交易，也可以稍后重试。

现在对于持有合法背书的交易，我们可以开始使用排序服务。提交客户端通过 broadcast (blob) 调用排序服务，这个时候 blob=endorsement。如果客户端没有能力直接调用排序服务，那么它可以通过它所选择的节点代理执行 broadcast(blob)，如此一个节点必须被客户端信任，并且不会从背书中删除任何消息，否则交易可能会被当作非法的。注意，即使如此，一个代理节点也不可能伪造一个合法的背书信息。

6.4.4 排序服务向所有节点投递交易消息

当发生一个 deliver 事件（seqno、prevhash、blob），并且节点已经对那些序列号低于 seqno 的 blob 大对象应用了所有的状态更新，此时节点会进行以下处理。

1）根据链代码（blob.tran-proposal.chaincodeID）的策略，节点检查 blob.endorsement 是否合法。

2）有没有违反核实依赖（blob.endorsement.tran-proposal.readset）。在更加复杂的用例中，背书信息的交易提案可能会不同，在这个情况下，背书策略指定了状态如何演化。根据状态更新选择的一致性内容（consistency property）或“隔离保证（isolation guarantee）”的不同，依赖验证有多种实现方式。串行化是一种默认的隔离保证，除非链代码背书策略制定了一个不同的依赖验证。比如，可串行性（serializability）要求每个 readset 和 writeset 里键对应的版本号必须与状态里面键的版本号相同，并抛弃掉不能满足这个要求的交易。

如果所有这些检查都能通过，那么就认为交易是合法的，然后提交交易。在这种情况下，节点在原始总账（peer ledger）的位掩码中为这个交易标记 1 然后应用 blob.tran-proposal.stateUpdates 到区块链状态，注意只有提交了的交易才可以改变状态。

如果 blob.endorsement 的背书策略核实失败，那么交易就是非法的，节点在原始总账的位掩码中用 0 标记交易。值得注意的是，非法的交易不会更改状态。

注意针对一个给定的序号，所有正常的节点在处理一个 deliver 事件（区块）之后，必须具有相同的状态。换句话说，通过排序服务的保证，所有正常的节点都将收到 deliver(seqno, prevhash, blob) 事件的相等的序号。

图 6-5 展示了一种可能的交易流程。

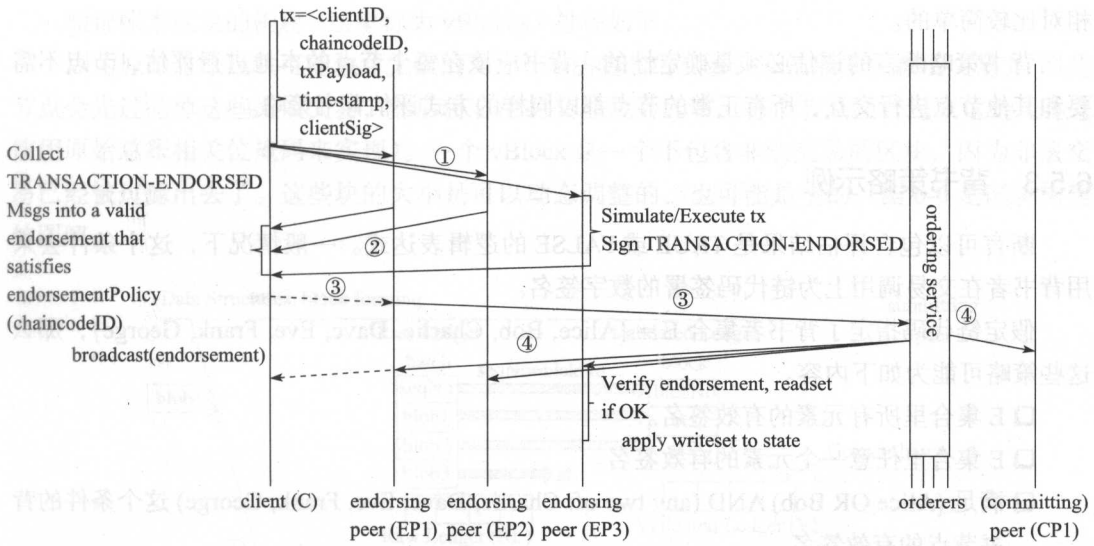


图 6-5　可能的交易流程

6.5　背书策略

6.5.1　背书策略规范

背书策略是对交易进行背书的条件。区块链节点有预先指定的背书策略集，这些背书策略集安装了特定链代码的部署交易指定。根据背书策略，只有经过背书处理的交易才是合法的交易。

6.5.2　交易评估与背书策略

只要根据策略进行了背书处理，就可以合法地声明交易。要实现链代码的调用交易，首先要获取满足链代码策略的背书，否则它是不允许提交的。这可以通过提交客户端和背书节点之间的交互来实现，关于这些内容可参考第 2 章的说明。

从形式上来说，背书策略是对背书和可能有的下一步状态的断言（判断是 TRUE 还是 FALSE）。部署交易的背书策略是从系统层面的策略来获取的（比如，从系统链码获取）

背书策略断言会涉及特定的变量。这些变量可以是如下几种。

- 链码相关的键或标识符（在链码的元数据里面能够找到），如背书节点集合。
- 更多的链代码元数据。
- 背书本身的元素。
- 更多的其他东西。

断言列表是由简单到丰富、由易到难的。也就是说，支持只有键和节点标识符的策略是

相对比较简单的。

背书策略断言的评估必须是确定性的。背书应该在每个节点的本地进行评估，节点不需要和其他节点进行交互，所有正常的节点都以同样的方式评估背书策略。

6.5.3 背书策略示例

断言可以包含评估结果是 TRUE 或 FALSE 的逻辑表达式。一般情况下，这个条件会采用背书者在交易调用上为链代码签署的数字签名。

假定链代码指定了背书者集合 E = {Alice, Bob, Charlie, Dave, Eve, Frank, George}，那么这些策略可能为如下内容。

- E 集合里所有元素的有效签名。
- E 集合里任意一个元素的有效签名。
- 满足 (Alice OR Bob) AND (any two of: Charlie, Dave, Eve, Frank, George) 这个条件的背书节点的有效签名。
- 7 个背书者中任意 5 个节点的有效签名（更通用的情况是，有 n > 3f 个背书节点的链码，需要 n 个背书节点中有 2f+1 个有效签名，或者任意一个超过 (n+f)/2 个背书节点的组有效签名）。
- 假定背书者都有一个投注或权重，像 {Alice=49, Bob=15, Charlie=15, Dave=10, Eve=7, Frank=3, George=1}，总的权重是 100，那么这个策略要求集合中拥有大多数权重的节点的有效签名（即一个总投注严格大于 50 的组），比如 {Alice, X}，其中 X 不等于 George 或 { 除开 Alice 的所有人 } 等这些都满足总投注大于 50。
- 权重的分配可以是静态的（固定在链代码的元数据中）也可以是动态的（依赖于链代码的状态，在执行期间可以修改）(比如之前示例中所展示的)。
- 在交易提案 1 上的有效签名（Alice OR Bob）和在交易提案 2 上的有效签名（any two of: Charlie, Dave, Eve, Frank, George），2 个交易提案的区别仅仅是在于它们的背书节点和状态更新。

这些策略能起到多大作用还要取决于应用，取决于背书节点失效或行为异常时所需要的解决方案的弹性，同时还取决于其他不同的属性。

6.6 验证总账（1.0 版本之后的功能）和原始总账检查点（精简）

6.6.1 验证总账

为维护包含有效和提交交易的账本抽象信息（比如比特币中），节点可能还会维护除了状态和总账之外的验证总账（VLedger）。验证总账是一条源自总账的 Hash 链，其过滤掉了无效的交易。

验证账本区块的构建（这里称为 vBlocks）过程如下。

因为原始总账区块可能包含无效的交易（非法签名的交易，非法版本依赖的交易），因此节点会先过滤掉这些无效交易，然后将合法的交易交给区块。每个节点独自完成这些（通过使用原始总账相关位掩码来实现）。一个 vBlock 是一个不包含非法交易的区块，因为非法交易已经被过滤出去了。这些块的大小是可以动态调整的，也可能是空的。图 6-6 是区块构建的图解。

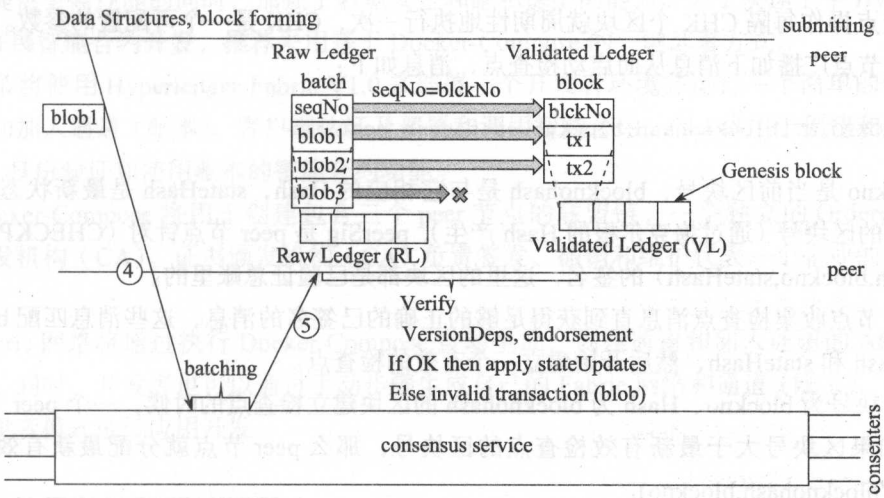


图 6-6　区块构建的图解

vBlocks 是每个节点形成的 Hash 链。更具体地说，一个验证总账的每个区块包含如下内容。

- 上一个区块的 Hash 值。
- 区块号。
- 从上一个区块形成之后节点提交的所有有效交易的有序列表（即，相应批块的有效交易列表）。
- 产生当前区块的相应批块的 Hash。

peer 节点会将所有的信息连接在一起并计算 Hash，以得出已验证总账里区块的 Hash。

6.6.2　原始总账检查点

总账（ledger）包含了非法交易，这些交易没有必要永久性地记录下来。尽管如此，节点也不能直接放弃原始总账区块，不过，一旦建立了相应的验证区块，就会删除原始总账。也就是说，在这种情况下，如果一个新的节点加入了网络，那么其他节点不可能将丢弃了的区块（保存在原始总账中的区块）转移到新加入的节点中，也不能让新加入的节点相信传输给它的（已验证）区块的有效性。

为了方便精简原始总账，本文描述了以下的检查点机制：这个机制通过节点网络来验证区块的有效性，允许建立了检查点的验证区块替换废弃的原始总账区块，从而逐步减少存储空间，因为不需要存储无效交易。此外，这个机制还可以减少重新构建新加入网络节点的工作量（因为当通过重播原始总账重新构建状态时，是不需要建立单个交易的有效性的，但是可以直接重播包含在验证节点中的状态更新）。

1. 检查点协议

检查点操作每隔 CHK 个区块就周期性地执行一次，CHK 是一个可配置的参数。节点通过向其他节点广播如下消息从而启动检查点，消息如下：

```
<CHECKPOINT,blocknohash,blockno,stateHash,peerSig>
```

blockno 是当前区块号，blocknohash 是与它相应的 Hash，stateHash 是最新状态的 Hash 基于区块的区块号（通过梅克尔根的 Hash 产生），peerSig 是 peer 节点针对（CHECKPOINT,blocknohash,blockno,stateHash）的签名，这里的区块都是已验证总账里的。

peer 节点收集检查点消息直到获得足够的正确的已签名的消息，这些消息匹配 blockno、blocknohash 和 stateHash，然后开始建立一个有效的检查点。

为区块号为 blockno、Hash 为 blocknohash 的区块建立检查点的时候，一个 peer 节点：

- 如果区块号大于最新有效检查点的区块号，那么 peer 节点就分配最新有效检查点=(blocknohash,blockno)。
- 将构成一个有效检查点的 peer 节点签名集合保存到 latestValidCheckpointProof。
- （可选）精简批块号小于等于 blockno 的原始总账（peer ledger）。

2. 有效的检查点

很显然，检查点协议抛出了下列问题：一个 peer 节点要在什么时候精简它的原始总账。多少个检查点消息才是足够多。这个通过检查点有效性策略的定义，至少有 2 种可能的途径，也可以一起使用。

- LCVP：Local(peer-specific) checkpoint validity policy，本地检查点有效性策略。在一个指定的 peer 节点上的本地策略可能指定这个 peer 节点信任的一些 peer 节点集合，这些被信任的 peer 节点的检查点信息足够创建一个有效性的检查点。例如，LCVP 在 peer 节点 Alice 处可能定义 Alice 需要接收来自 Bob 的检查点信息，或者来自 Charlie 和 Dave 的检查点信息。
- GCVP：Global checkpoint validity policy，全局检查点有效性策略。检查点有效性策略可以全局指定。这个与本地 peer 策略类似，区别是他被系统级（区块链）粒度操作而不是 peer 节点粒度操纵。例如，GCVP 可能指定：
  - 每个 peer 节点可能信任一个检查点，这个检查点将通过 11 个不同的 peer 节点来确认。

- 有这么一个部署环境，每个投票节点都是一个 peer 节点，有 f 个投票节点可能是（拜占庭）故障的，每个节点可以信任由 f+1 个不同 Peer 节点确认过的检查点。

6.7　Fabric V1.0 开发者快速入门

Hyperledger Fabric 1.0 版本重新设计了架构，新的设计可以实现更好的扩展性和安全性，在提供更高性能的同时，加强了数据安全和隐私保护的能力。为了快速上手 Hyperledger Fabric 开展智能合约开发，推荐采用基于 Docker-Compose 的一键部署方式。

本节将使用 Hyperledger Fabric V1.0 来部署一个开发者环境并运行一个简单的例子，包括创建和加入通道（账本）、客户端认证及部署和调用智能合约。CLI 将用于创建和加入通道（账本）、身份验证和使用账本的智能合约功能。

Docker-Compose 将用于创建包含三个 peer 节点的联盟链、一个独立的 Orderer 和一个证书颁发机构（CA）。证书颁发机构（CA）负责签发、撤销和维护代表一个企业的加密认证要素。

Fabric 网络将通过执行 Docker-Compose 自动生成，创建通道和加入通道的 API 将会自动调用；同时，开发者也可以通过手动步骤生成自己的 Fabric 网络和通道（账本），或者直接使用开发者模式进行应用开发。

6.7.1　前置条件和系统配置

- Docker：采用 v1.12 及更高版本。
- Docker-Compose：采用 v1.8 及更高版本。

1. 安装 Docker

Docker 支持 Linux 常见的发行版本，如 Redhat/CentOS/Ubuntu 等，建议使用 Docker 1.12 及更高的版本。这里以 CentOS 7.2 为例给出安装命令：

```
$ sudo yum-config-manager \
--add-repo \
```

https://docs.docker.com/engine/installation/linux/repo_files/centos/docker.repo

```
$ sudo yum makecache fast
$ sudo yum -y install docker
```

2. 安装 Docker-Compose

首先，安装 python-pip 软件包：

```
$ sudo yum install python-pip
```

然后，安装 Docker-Compose，建议为 1.8.0 及更高的版本：

```
$ sudo pip install --upgrade pip

$ sudo pip install docker-compose
```

6.7.2 下载源代码，创建 Fabric 网络

1）获取 Fabric 代码，命令如下：

```
mkdir -p $GOPATH/src/github.com/hyperledger
cd $GOPATH/src/github.com/hyperledger
git clone https://github.com/hyperledger/fabric.git
```

2）然后编译 configtxgen 工具：

```
cd $GOPATH/src/github.com/hyperledger/fabric
make configtxgen
```

3）拉取 Fabric1.0 镜像，并设置标签：

```
docker pull hyperledger/fabric-orderer:x86_64-1.0.0-alpha
docker pull hyperledger/fabric-peer:x86_64-1.0.0-alpha
docker pull hyperledger/fabric-zookeeper:x86_64-1.0.0-alpha
docker pull hyperledger/fabric-couchdb:x86_64-1.0.0-alpha
docker pull hyperledger/fabric-kafka:x86_64-1.0.0-alpha
docker pull hyperledger/fabric-ca:x86_64-1.0.0-alpha
docker pull hyperledger/fabric-ccenv:x86_64-1.0.0-alpha
docker pull hyperledger/fabric-javaenv:x86_64-1.0.0-alpha
docker tag hyperledger/fabric-orderer:x86_64-1.0.0-alpha hyperledger/fabric-orderer:latest
docker tag hyperledger/fabric-peer:x86_64-1.0.0-alpha hyperledger/fabric-peer:latest
docker tag hyperledger/fabric-zookeeper:x86_64-1.0.0-alpha hyperledger/fabric-zookeeper:latest
docker tag hyperledger/fabric-couchdb:x86_64-1.0.0-alpha hyperledger/fabric-couchdb:latest
docker tag hyperledger/fabric-kafka:x86_64-1.0.0-alpha hyperledger/fabric-kafka:latest
docker tag hyperledger/fabric-ca:x86_64-1.0.0-alpha hyperledger/fabric-ca:latest
docker tag hyperledger/fabric-ccenv:x86_64-1.0.0-alpha hyperledger/fabric-ccenv:latest
docker tag hyperledger/fabric-javaenv:x86_64-1.0.0-alpha hyperledger/fabric-javaenv:latest
```

6.7.3 生成配置文件

configtxgen 工具用于生成两个要素：1）orderer 的创始区块；2）配置 Fabric 通道的配置文件。

orderer block 是用于 order 排序服务的创始区块，在创建通道的时候将被广播到 order 排序服务。

configtx.yaml 包含示例区块链网络内角色的定义；/crypto 目录包含管理员证书、CA 证书、每个角色的私钥和用于签名的证书

1）执行生成配置脚本，命令如下：

```
cd $GOPATH/src/github.com/hyperledger/fabric/examples/e2e_cli
./generateCfgTrx.sh <channel-ID>
```

你可以设置 channel-ID 为你的通道名称，默认名称为 mychannel。

2）执行脚本后，可以在终端上看到如下输出：

```
Generating new channel configtx
Creating no-op MSP instance
Obtaining default signing identity
Creating no-op signing identity instance
Serializing identity
signing message
signing message
Writing new channel tx
```

这样，创建区块链网络的必要准备工作就做完了。

6.7.4　使用 Docker 创建 Fabric 网络 & 创建 / 加入通道（账本）

1）使用 Docker-Compose 启动 Fabric 集群并执行测试，命令如下：

```
[CHANNEL_NAME=<channel-id>] docker-compose up -d
```

2）查看你的容器，命令如下：

```
docker ps
```

终端执行 docker ps 显示第一阶段启动了 Fabric 集群，第二阶段部署了智能合约 mycc 并运行执行结束，应该存在一个 Fabric 网络和一个通道（账本），通道包含 4 个节点，分别是 peer0、Peer1、Peer2 和 Peer3，并且在 Peer0、Peer2、Peer3 上已经安装了智能合约 mycc（example-chaincode02）。

3）验证是否成功创建了创世块，命令如下：

```
docker exec -it cli bash
ls -ltr <channel-ID>.block
```

6.7.5　示例合约执行过程解析

1）CLI 容器内执行了一个脚本 - script.sh。脚本首先通过获取指定的通道名称执行 createChannel 命令。

2）createChannel 执行完成的输出是 Order 的创始区块 - <channel-ID>.block。

3）向四个 Peer 节点发送创始区块并执行 joinChannel 命令。

4）至此，创建了一个由四个 Peer 节点和两个组织（Org0、Org1）组成的通道。

5）Org 关系都在 configtx.yaml 中进行定义。

6）接下来执行智能合约 -chaincode_example02 被部署到 PEER0 和 PEER2。

7）部署在 PEER2 上的合约代码会执行实例化，实例化是指启动合约容器并初始化与合约相关联的 key/value，示例里启动了一个 dev-peer2-mycc-1.0 的容器。

8）实例化也会传递有关的背书策略，这里背书策略的定义为 -P“OR('Org0MSP.member', 'Org1MSP.member')”，意思是任何交易都必须由绑定到 Org0 或 Org1 的节点来实施签名背书，交易才有效。

9）对 PEER0 发出“a”的 query 交易。合约之前已经被部署在 PEER0 上，因此会启动 dev-peer0-mycc-1.0 容器并返回查询的结果。

10）向 PEER0 发起 invoke 交易，从 a 到 b 转移 10 个单位。

11）将智能合约代码部署到 PEER3 上。

12）向 PEER3 发送“a”的 query 交易，再次启动 dev-peer3-mycc-1.0 为名称的第三个合约容器并返回查询值 90，正确反映之前的交易过程，即 a 的值由 100 修改为 90。

6.7.6 查看智能合约执行日志

通过合约所在的容器查看有关日志，代码如下：

```
$ docker logs dev-peer2-mycc-1.0
04:30:45.947 [BCCSP_FACTORY] DEBU : Initialize BCCSP [SW]
ex02 Init
Aval = 100, Bval = 200
# 查看 Peer2 节点的 mycc 合约执行日志
$ docker logs dev-peer0-mycc-1.0
04:31:10.569 [BCCSP_FACTORY] DEBU : Initialize BCCSP [SW]
ex02 Invoke
Query Response:{"Name":"a","Amount":"100"}
ex02 Invoke
Aval = 90, Bval = 210

$ docker logs dev-peer3-mycc-1.0
04:31:30.420 [BCCSP_FACTORY] DEBU : Initialize BCCSP [SW]
ex02 Invoke
Query Response:{"Name":"a","Amount":"90"}
```

6.7.7 手工创建和加入通道

为了在 CLI 容器里手动执行创建通道和加入通道 API，我们需要编辑 Docker-Compose 文件。首先，用任意文本编辑器打开 docker-compose.yaml 注释掉 ./scripts/script.sh 脚本并去除 /bin/bash 前的注释，编辑操作如下：

```
cli:
  container_name: cli
  <CONTENT REMOVED FOR BREVITY>
  working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
#command: /bin/bash -c './scripts/script.sh ${CHANNEL_NAME}; '
command: /bin/bash
```

然后正式手动执行创建通道和加入通道 API。

1）进入 cli 容器，命令如下：

```
docker exec -it cli bash
```

2）向 Orderer 发送 createChannel API，命令如下：

```
peer channel create -o orderer0:7050 -c mychannel -f crypto/orderer/channel.tx
--tls $CORE_PEER_TLS_ENABLED --cafile /opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/orderer/localMspConfig/cacerts/ordererOrg0.pem
```

3）向 Peer0 发送 joinchannel API，命令如下：

```
peer channel join -b mychannel.block
```

CreateChannel 执行完毕之后，将返回一个创世区块（mychannel.block），然后可以执行加入通道的指令，将 Genesis block 作为参数向 peer0 发送 joinchannel API。

需要说明的是，通道的定义都保存在创世块内。

4）通过修改环境变量，只需向 peer1 或 Peer2 重新发送上述命令即可将全部 Peer 节点加入通道，如下是向 Peer1 发送命令的格式：

```
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peer/peer1/localMspConfig
CORE_PEER_ADDRESS=peer1:7051
CORE_PEER_LOCALMSPID="Org0MSP"
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peer/peer1/localMspConfig/cacerts/peerOrg0.pem
peer channel join -b mychannel.block
```

一旦全部 Peer 节点都加入了通道，你就可以查询账本而无需在每个节点都部署智能合约。

6.7.8 使用命令行工具部署、调用、查询智能合约

（1）运行部署命令，包括安装步骤和初始化步骤

这两条命令将在通道 mychannel 上的 Peer0 节点上安装一个名为 MYCC 的智能合约，初始化智能合约实例将 A 和 B 的值分别初始化为 100 和 200：

```
peer chaincode install -n mycc -v 1.0 -p github.com/hyperledger/fabric/examples/
chaincode/go/chaincode_example02
peer chaincode instantiate -o orderer0:7050 --tls $CORE_PEER_TLS_ENABLED
--cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/orderer/local-
MspConfig/cacerts/ordererOrg0.pem -C mychannel -n mycc -v 1.0 -p github.com/
hyperledger/fabric/examples/chaincode/go/chaincode_example02 -c '{"Args":["init",
"a", "100", "b","200"]}' -P "OR ('Org0MSP.member','Org1MSP.member')"
```

此时系统会生成类似 dev-peer0-mycc-<chaincodeid> 的容器镜像，主要用于智能合约的代码执行。

（2）运行调用命令

这个命令是从 A 移动 10 个单位到 B，示例代码如下：

```
peer chaincode instantiate -o orderer0:7050 --tls $CORE_PEER_TLS_ENABLED
--cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/orderer/local-
MspConfig/cacerts/ordererOrg0.pem -C mychannel -n mycc -v 1.0 -p github.com/
hyperledger/fabric/examples/chaincode/go/chaincode_example02 -c '{"Args":
["init","a", "100", "b","200"]}' -P "OR ('Org0MSP.member','Org1MSP.member')"
```

（3）运行查询命令

按照预期，查询 a 的返回值应该是 90，命令如下：

```
peer chaincode query -C mychannel -n mycc -c '{"Args":["query","a"]}'
```

终端显示结果如下：

```
Query Result: 90
```

你可以在任何时间发出 exit 命令退出 CLI 容器。

6.7.9 开发环境故障排除

- 如果已有容器在运行，则执行 Docker-Compose 命令可能会收到报错，显示端口已被占用。如果发生了这种情况，则需要杀死使用该端口的容器。
- 如果发现缺失了部分文件，请确保 Curl 命令成功执行，并确保已经通过 cd 命令找到了代码下载的目录。
- 确保在每次运行后清除有关文件
- 如果编译 Docker 出现报错，则可以删除镜像并从头开始：

```
make clean
make peer-docker orderer-docker
```

- 如果出现下面的报错：

```
Error: Error endorsing chaincode: rpc error: code = 2 desc = Error installing
chaincode code mycc:1.0(chaincode /var/hyperledger/production/chaincodes/mycc.1.0 exits)
```

则其可能是残留了先前运行的合约容器镜像（例如 peer0-peer0-mycc-1.0 或 peer1-peer0-mycc1-1.0），删除以后再试一下：

```
docker rmi -f $(docker images | grep peer[0-9]-peer[0-9] | awk '{print $3}')
```

- 清理整个区块链网络，使用脚本的 down 选项：

```
./network_setup.sh down
```

6.7.10 Fabric 常用的 Docker 命令

删除一个容器的命令如下：

```
docker rm <containerID>
```

强制删除一个容器的命令如下：

```
docker rm -f <containerID>
```

强制删除全部容器的命令如下：

```
docker rm -f $(docker ps -aq)
```

删除一个镜像的命令如下：

```
docker rmi <imageID>
```

强制删除一个镜像的命令如下：

```
docker rmi -f <imageID>
```

强制删除全部镜像的命令如下：

```
docker rmi -f $(docker images -q)
```

6.8　智能合约开发

6.8.1　智能合约的定义

chaincode 是部署在 Hyperledger Fabric 网络节点上，可用来与分布式账本进行交互的一段程序代码，也称为狭义范畴上的“智能合约”。chaincode 在验证节点上的隔离沙盒（目前称为 Docker 容器）中执行，并通过 gRPC 协议来被相应的验证节点或客户端调用和查询。

Hyperledger 支持多种计算机语言实现的 chaincode，包括 Golang、JavaScript、Java 等。下面将分别以 Go、Java 语言为例来介绍智能合约开发，并对案例代码进行解释说明。

6.8.2　GO 语言智能合约的开发和部署

1. 实现智能合约的接口

Go 语言主要依赖 chaincode 的 shim 接口来实现核心业务逻辑。Fabric1.0 的 shim 接口主要包含两个核心的函数，分别是 init 和 invoke。功能函数都以函数名和字符串结构作为输入，主要的区别在于函数的功能用途。

（1）init() 函数

当首次部署 chaincode 代码时，init 函数会被调用。如同名字所描述的，该函数用来做一些初始化的工作。

（2）invoke() 函数

当通过调用 chaincode 代码来做一些实际性的工作时，可以使用 invoke 函数。发起的交易将会被链上的区块获取并记录。

它以被调用的函数名作为参数，并基于该参数去调用 chaincode 中匹配的的 go 函数。

Fabric1.0 会将 Query() 函数整合进 invoke 函数，以便进行灵活地查询和调用。

（3）main() 函数

最后，需要创建一个 main 函数，当在每个节点中部署 chaincode 的实例时，会调用该函数，不过，仅仅在 chaincode 在某节点上注册时才调用。

2. 智能合约的依赖关系

chaincode 需要引入如下的软件包。

- fmt：包含了 println 等标准函数。
- errors：为标准的 errors 类型包。
- github.com/hyperledger/fabric/core/chaincode/shim：与 chaincode 节点交互的接口代码。shim 包提供了 stub.PutState 与 stub.GetState 等方法来写入和查询区块链上 key/value 的状态。核心的 shim 包，通过封装 gRPC 消息的方式到验证节点来完成操作。

3. 智能合约的数据格式

在 Fabric 0.6 中，数据持久化存储采用 rocksdb，除了基本的 key-value 格式的数据支持之外，智能合约的 GO 接口支持以 Table 的形式定义合约中的数据，但是 Fabric 1.0 中移除了 Table 相关的接口。

在 Fabric 1.0 中，建议基于 JSON 格式构造智能合约数据，在例子 marbles02 中，可通过如下代码定义结构体 marble 为 JSON 格式：

```
type marble struct {
    ObjectType      string `json:"docType"`
    Name            string `json:"name"`
    Color           string `json:"color"`
    Size            int    `json:"size"`
    Owner           string `json:"owner"`
}
```

该结构体定义了 marble 的类型、名称、颜色、尺寸、拥有者等属性。

下面是 Table 实现方式和 JSON 实现方式的一些对比。

基于 Table 的实现方式	基于 JSON 的实现方式
❑ 关系型数据库的方式 ❑ 需要事先定义好表结构 ❑ 在之后合约的升级过程中很难更改表结构 ❑ 不能支持分层数据 ❑ 和账本的底层数据不一致 ❑ 该方式无法满足 Fabric 的功能 • 例如无法像预期的那样随意按列查询数据 ❑ 代码结构更复杂，导致合约代码也复杂	❑ NoSQL 的方式 :key/value (LevelDB), document db (CouchDB) ❑ 不需要事先定义好表结构 ❑ 在之后的合约升级过程中可以很容易地添加新的 JSON 域 ❑ 支持分层数据 ❑ 和账本的底层数据一致 ❑ 该方式可以满足 Fabric 的功能 • 查询基于 key 或 partial key range ❑ 更少的代码量，采用内置结构 JSON marshaling ❑ 更加适合下一代的账本功能要求 • 可以以任何字段查询账本，无论是在合约内还是合约外 • 在使用基于 JSON 的数据库时更加有效 (CouchDB)

4. 智能合约的接口解析

在 Fabric 1.0 中，移除了 Table 相关的所有读写接口，同时，还提供了如下读写接口供合约调用。

- GetState(key string) ([]byte, error)：按照给定的 key 查询返回对应值的 byte 数组。
- PutState(key string, value []byte) error：将给定的 key 和 value 写入账本。
- DelState(key string) error：在账本中移除给定的 key 和对应的 value 记录。
- CreateCompositeKey(objectType string, attributes []string) (string, error)：该接口会构建一个由 objectType 和 attributes 组成的组合 key，其中 objectType 和 attributes 必须是有效的 utf8 字符串且不能包含 string(0) 和 string(utf8.MaxRune)。
- SplitCompositeKey(compositeKey string) (string, []string, error)：该接口会将构建好的组合 key 重新按照组合方式拆分成 objectType 和 attributes。
- PartialCompositeKeyQuery(objectType string, keys []string) (StateQueryIteratorInterface, error)：在组合键中查询包含给定 objectType 和 keys 的组合键，返回这些组合键的迭代器，注意 keys 必须是组合键的一部分，如果是全部组合键的话会返回空的迭代器。
- RangeQueryState(startKey, endKey string) (StateQueryIteratorInterface, error)：在账本中查询 key 值在 startKey 和 endKey 之间的记录的迭代器，迭代器中 key 的顺序是随机的。
- GetQueryResult(query string) (StateQueryIteratorInterface, error)：该接口会在状态数据库（如 CouchDB）中执行 rich query，实现对任意字段的查询，参数 query 是底层状态数据库的查询语法，该接口仅支持使用状态数据库（如 CouchDB）的情况。返回一个包含所需记录的迭代器。

5. 智能合约案例代码分析

本智能合约案例基于 Fabric 1.0 来实现弹珠的生成、转让和查询，目前 Fabric 1.0 将 query 方法合并至 invoke 接口，统一了使用方式。

（1）将 JSON 数据写入账本

可通过如下命令将指定的 key 和 JSON 数据写入账本：

```
1        // ==== Create marble object and marshal to JSON ====
2        objectType := "marble"
3        marble := &marble{objectType, marbleName, color, size, owner}
4        marbleJSONasBytes, err := json.Marshal(marble)
5        if err != nil {
6              return shim.Error(err.Error())
7        }
8        //=== Save marble to state ===
9        err = stub.PutState(marbleName, marbleJSONasBytes)
10        if err != nil {
11              return shim.Error(err.Error())
12         }
```

其中，第3行实现了定义的JSON结构体，第4行以byte数组的形式存储JSON数据，第9行调用Putstate接口，将name作为key，JSON数据作为value写入账本。

（2）按照查询需求创建索引（不使用CouchDB等状态数据库）

在本例中，如果有按照颜色查询marble的需求，则需按如下方法建立索引：

```
1        //Index the marble to enable color-based range queries
2        //An 'index' is a normal key/value entry in state.
3     //The key is a composite key, with the elements that you want to range
query on listed first.
4        //In our case, the composite key is based on indexName~color~name.
5     //This will enable very efficient state range queries based on composite
keys matching indexName~color~*
6        indexName := "color~name"
7        colorNameIndexKey, err :=
stub.CreateCompositeKey(indexName, []string{marble.Color, marble.Name})
8        if err != nil {
9              return shim.Error(err.Error())
10       }
11    //Save index entry to state. Only the key name is needed, no need to store
a duplicate copy of the marble.
12    //Note - passing a 'nil' value will effectively delete the key from state,
therefore we pass null character as value
13       value := []byte{0x00}
14       stub.PutState(colorNameIndexKey, value)
```

其中，第7行创建了由color和name组成的组合键，建立了color和name（主键）之间的索引关系，索引以名称“color~name”区别于其他索引。第14行将这个组合键作为key，一个空character作为value写入账本，索引信息在key中。

（3）按照颜色查询（不使用CouchDB等状态数据库）

查询命令如下：

```
1        // Query the color~name index by color
2        //This will execute a key range query on all keys starting with 'color'
3        coloredMarbleResultsIterator, err :=
stub.PartialCompositeKeyQuery("color~name", []string{color})
4        if err != nil {
5              return shim.Error(err.Error())
6        }
7        defer coloredMarbleResultsIterator.Close()

8        //Iterate through result set and for each marble found, transfer to newOwner
9        var i int
10       for i = 0; coloredMarbleResultsIterator.HasNext(); i++ {
11             //Note that we don't get the value (2nd return variable), we'll
               just get the marble name from the composite key
12             colorNameKey, _, err := coloredMarbleResultsIterator.Next()
13             if err != nil {
14                   return shim.Error(err.Error())
```

```
15              }

16              //get the color and name from color~name composite key
17              objectType, compositeKeyParts, err :=
stub.SplitCompositeKey(colorNameKey)
18              if err != nil {
19                      return shim.Error(err.Error())
20              }
21              returnedColor := compositeKeyParts[0]
22              returnedMarbleName := compositeKeyParts[1]
23              fmt.Printf("- found a marble from index:%s color:%s name:%s\n",
                objectType, returnedColor, returnedMarbleName)

24              //Now call the transfer function for the found marble.
25              //Re-use the same function that is used to transfer individual marbles
26              marbleAsBytes, err := stub.GetState(returnedMarbleName)
```

其中，第 3 行调用 PartialCompositeKeyQuery 接口查出包含符合颜色要求的组合键的迭代器（该函数调用了 RangeQueryState 接口），第 10 行对迭代器做循环，取出符合要求的组合键，第 17 行用 SplitCompositeKey 接口解析出 marble 的 name（主键），第 26 行用取出的 name（主键）查出符合颜色要求的 Marble 的 JSON 数据。

（4）按照拥有者查询（使用 CouchDB 等状态数据库）

使用 CouchDB 的 rich query 功能按照拥有者进行查询，该情况下无需在合约里额外建立任何索引信息，可以直接用 CouchDB 的查询语法进行任意查询：

```
1       // queryMarblesByOwner queries for marbles based on a passed in owner.
2    //This is an example of a parameterized query where the query logic is baked
into the chaincode, and accepting a single query parameter (owner).
3       //Only available on state databases that support rich query (e.g. CouchDB)
4       func (t *SimpleChaincode) queryMarblesByOwner
(stub shim.ChaincodeStubInterface, args []string) pb.Response {
5               if len(args) < 1 {
6                       return shim.Error("Incorrect number of arguments. Expecting 1")
7               }

8               owner := strings.ToLower(args[0])

9               queryString :=fmt.Sprintf
("{\"selector\":{\"docType\":\"marble\",\"owner\":\"%s\"}}", owner)

10              queryResults, err := getQueryResultForQueryString(stub, queryString)
11              if err != nil {
12                      return shim.Error(err.Error())
13              }
14              return shim.Success(queryResults)
15      }
```

其中，第 10 行的 getQueryResultForQueryString 函数调用了第 3 节的 GetQueryResult 接口，

循环返回的迭代器，将查询出的 key 和 value 构建为 JSON 数组，最终以 byte 数组的形式返回：

```
func getQueryResultForQueryString
(stub shim.ChaincodeStubInterface, queryString string) ([]byte, error) {

2           fmt.Printf
("- getQueryResultForQueryString queryString:\n%s\n", queryString)

3           resultsIterator, err := stub.GetQueryResult(queryString)
4           if err != nil {
5               return nil, err
6           }
7           defer resultsIterator.Close()

8           //buffer is a JSON array containing QueryRecords
9           var buffer bytes.Buffer
10          buffer.WriteString("[")

11          bArrayMemberAlreadyWritten := false
12          for resultsIterator.HasNext() {
13              queryResultKey, queryResultRecord, err := resultsIterator.Next()
14              if err != nil {
15                  return nil, err
16              }
17  //Add a comma before array members, suppress it for the first array member
18              if  bArrayMemberAlreadyWritten == true {
19                  buffer.WriteString(",")
20              }
21              buffer.WriteString("{\"Key\":")
22              buffer.WriteString("\"")
23              buffer.WriteString(queryResultKey)
24              buffer.WriteString("\"")

25              buffer.WriteString(", \"Record\":")
26              //Record is a JSON object, so we write as-is
27              buffer.WriteString(string(queryResultRecord))
28              buffer.WriteString("}")
29              bArrayMemberAlreadyWritten = true
30          }
31          buffer.WriteString("]")

32          fmt.Printf
("- getQueryResultForQueryString queryResult:\n%s\n", buffer.String())

33          return buffer.Bytes(), nil
34      }
```

6.8.3 Java 智能合约的编写与部署

目前 Fabric 1.0 基础框架中已经定义了良好的 Java 接口，供 Java 程序员采用 Java 实现

智能合约。

基于 Fabric 实现一个 Java 智能合约的主要步骤具体如下。

1）定义智能合约需要实现的功能及账本中需要保存的数据格式。

2）实现智能合约类，并且使其继承自 ChaincodeBase。

3）定义智能合约需要支持的操作，并在智能合约类的 run 接口中实现这些操作。

4）在类中实现 main 函数。

下面就来详细解释如何实现以上步骤。

1. 定义智能合约需要实现的功能

我们通过智能合约实现最简单的 key-value 服务，此服务中提供了两个操作，即 Get 和 Put。对于 Get 操作，用户需要提供 key，智能合约返回对应的 value。对于 Put 操作，用户提供 key/value，智能合约将此 key/value 键值对保存到账本中。

2. 实现智能合约类

Java 智能合约类的定义如下：

```
package example
import org.hyperledger.java.shim.ChaincodeBase;
import org.hyperledger.java.shim.ChaincodeStub;
public class Example extends ChaincodeBase {
  ...
}
```

3. 实现智能合约业务逻辑

基于 Fabric 1.0 的 Java 智能合约，由于其继承自 ChaincodeBase 类，因此需要在 run 处理函数中定义此智能合约的业务逻辑。在 ChaincodeBase 中，run 接口的定义如下：

```
public abstract String run(ChaincodeStub stub, String function, String[] args);
```

其中，各个参数的定义如下。

- stub：Fabric 基础框架提供的账本操作接口类。
- function：用户调用智能合约时所指定的调用操作。
- args：用户调用智能合约时所提供的调用参数。

在示例代码中智能合约的具体实现如下：

```
1 @override
2 public String run(ChaincodeStub stub, String function, String[] args) {
3     switch (function) {
4     case "get" :
5         key = args[0];
6         return stub.getState(key);
7     case "put" :
8         key = args[0]
9         value = args[1]
```

```
10          return stub.putState(key);
11      default:
12          log.error( "unknown function: " + function);
13      }
14      return null;
15  }
```

在上述示例代码中，我们根据用户的操作，分别调用 ChaincodeStub 的 getState 和 putState 接口为用户提供 key-value 操作。

4. 实现智能合约的 main 函数

可在智能合约的 main 函数中完成智能合约线程的启动，代码如下所示：

```
1   public static void main(String[] args) throws Exception {
2       new Example().start(args);
3   }
```

其中 Example 调用其基类 ChaincodeBase 的 start 函数，启动智能合约的请求处理线程，此线程将建立与 Fabric 集群中其他节点的链接，并监听此链接接收请求，调用智能合约的 run 函数处理请求，并返回处理的结果。

5. 智能合约的账本存取接口

从上面示例中可以看出，在 Fabric 中实现智能合约类主要依赖于 Fabric 框架中提供的 ChaincodeBase 类和 ChaincodeStub 类。其中 ChaincodeBase 类作为智能合约类的父类，为智能合约提供了与 Fabric 其他节点建立链接、接收用户请求、返回请求结果等基础框架的功能；而 ChaincodeStub 类为智能合约提供了操作智能合约账本的功能。

ChaincodeStub 类为智能合约提供了基本的 key/value 接口，以及简单的查询接口。对各个操作接口的功能介绍如下。

- public String getState(String key)：按照给定的 key 查询返回对应值的 byte 数组。
- public void putState(String key, String value)：将给定的 key 和 value 写入账本。
- public void delState(String key)：在账本中移除给定的 key 和对应的 value 记录。
- public Map<String, String> partialCompositeKeyQuery(String objectType, String[] attributes)：首先基于 objectType 和 attributes 构建一个组合前缀键，然后利用此组合前缀键在账本中查询所有满足此前缀的纪录。
- public Map<String, String> rangeQueryState(String startKey, String endKey)：在账本中查询 key 值在 startKey 和 endKey 之间的记录。

从上面的几个接口可以看出，目前 Fabric 1.0 的账本接口将各种数据结构通过 String 的形式来表现，因此目前 Fabric 也提倡使用 JSON 的形式组织智能合约需要的数据记录。

6. 智能合约案例代码分析

参考 Fabric 1.0 的框架方法，现在按照前面介绍的步骤实现 Fabric marble 智能合约的

Java 版本。

（1）智能合约功能定义

在开始写代码之前，先定义此智能合约的功能。将其定义为弹珠管理系统，每个弹珠都有一个唯一的名字，并且具有以下几个属性。

- 所有者：owner
- 颜色：color
- 大小：size

因此我们将每个弹珠在账本中的 JSON 表示定义为：

```
{
    "name": "marble-name",
    "owner": "marble-owner",
    "color": "marble-color",
    "size": "marble-size"
}
```

（2）实现智能合约类

定义智能合约类为 MarbleExample，继承自 ChaincodeBase，并实现其 run 接口：

```
public class MarbleExample extends ChaincodeBase {
    private static Log log = LogFactory.getLog(MarbleExample.class);

    @java.lang.Override
    public String run(ChaincodeStub stub, String function, String[] args) {
        switch (function) {
                ….
```

（3）实现弹珠管理的业务逻辑

- 在此智能合约中我们将实现如下功能。
- 创建弹珠
- 转让弹珠
- 按弹珠所有人对弹珠进行查询
- 删除弹珠

这些功能都需要在 MarbleExample 的 run 函数中来实现。

- 创建弹珠

如下代码实现了创建弹珠的功能：

```
            case "createMarble":
                //args[0]: name
                //args[1]: owner
                //args[2]: color
                //args[3]: size
                String marbleName = args[0];
                String owner = args[1];
```

```
14              String color = args[2];
15              String size = args[3];
16
17              //construct json
18              JSONObject marbleObj = new JSONObject();
19              marbleObj.put("name", marbleName);
20              marbleObj.put("owner", owner);
21              marbleObj.put("color", color);
22              marbleObj.put("size", size);
23
24              //add marble
25              String marbleJsonStr = marbleObj.toJSONString();
26              stub.putState(marbleName, marbleJsonStr);
27
28              //add owner-marble index
29              String indexName = "owner~name"
30              String compKey = stub.createCompositeKey(indexName, {owner, marbleName});
31              stub.putState(compKey, marbleJsonStr);
32              break;
```

其中，第 7 ~ 15 行从调用参数中取出要创建的弹珠的各个参数；第 17 ~ 22 行根据输入的参数构建弹珠的 JSON 结构；第 24 ~ 26 行以弹珠名字作为 key，将弹珠加入到账本中；第 28 ~ 31 行为此弹珠构建了一个“owner ~ name”，用于后面实现按弹珠所有人对弹珠进行查询。

❑ 转让弹珠

如下代码实现了转让弹珠的功能：

```
34          case "transferMarble":
35              //args[0]: marble name
36              //args[1]: new owner of the marble
37              String marbleName = args[0];
38              String newOwner = args[1];
39
40              //get marble from ledger
41              String marbleJsonStr = stub.getState(marbleName);
42              JSONParser parser = new JSONParser();
43              JSONObject marbleObj = (JSONObject)parser.parse(new StringReader
(marbleJsonStr));
44              String oldOwner = marbleObj.get("owner");
45
46              //delete old owner-marble index
47              String indexName = "owner~name"
48              String compKey = stub.createCompositeKey(indexName, {oldOwner, marbleName});
49              stub.delState(compKey);
50
51              //change marble owner
52              marbleObj.put("owner", newOwner);
53
54              //update marble
```

```
55              String marbleJsonStr = marbleObj.toJSONString();
56              stub.putState(marbleName, marbleJsonStr);
57
58              //update owner-marble index
59              compKey = stub.createCompositeKey(indexName, {newOwner, marbleName});
60              stub.putState(compKey, marbleJsonStr);
61              break;
```

其中，第 35 ~ 38 行从调用参数中取出待转让的弹珠的名字，以及新的弹珠所有人；第 40 ~ 44 行从账本中取出原弹珠的 JSON 表示，并将其解析为 JSONObject；第 46 ~ 49 行删除原弹珠所有人与弹珠间的映射关系；第 51 行和第 52 行更改弹珠的所有人；第 54 ~ 56 行将更新的弹珠信息保存到 Fabric 账本中；第 58 ~ 60 行更新弹珠新的所有人与弹珠的映射关系。

❑ 按弹珠的所有人对弹珠进行查询

如下代码实现了弹珠查询的功能：

```
63          case "queryMarbleByOwner":
64              //args[0]: marble owner
65              String owner = args[0];
66
67              //query with owner-marble index
68              String indexName = "owner~name"
69              Map<String, String> marbles = stub.partialCompositeKeyQuery(inde
xName, {owner});
70
71              JSONArray marbleList = new JSONArray();
72              Iterator<String> iter = marbles.keySet().iterator();
73              while (iter.hasNext()) {
74                  String key = iter.Next();
75                  String marbleJsonStr = marbles.get(key);
76                  marbleList.add(marbleJsonStr);
77              }
78              result = marbleList.toJSONString();
79              break;
```

其中第 65 行会从调用参数中取出查询的所有人的名字；第 67 ~ 69 行利用 ChaincodeStub 提供的 partialCompositeKeyQuery 接口，查询此 owner 所拥有的所有弹珠；第 71 ~ 77 行将查询到的所有弹珠的 JSON 表示，保存到 JSON 数组中；第 78 行将 JSON 数组转换为字符串，返回给智能合约调用者。

❑ 删除弹珠

如下代码实现了删除弹珠的功能：

```
81          case "deleteMarble":
82              //args[0]: marble name
83              String marbleName = args[0];
84
85              //get marble from ledger
```

```
86              String marbleJsonStr = stub.getState(marbleName);
87              JSONParser parser = new JSONParser();
88              JSONObject marbleObj = (JSONObject)parser.parse(new StringReader
(marbleJsonStr));
89
90              //delete owner-marble index
91              String owner = marbleObj.get("owner");
92              String indexName = "owner~name"
93              String compKey = stub.createCompositeKey(indexName, {owner, marbleName});
94              stub.delState(compKey);
95
96              //delete marble
97              stub.delState(marbleName);
98              break;
```

其中，第 82 行和第 83 行会从调用参数中查询要删除的弹珠的名字；第 85 ~ 88 行从 Fabric 账本中取出对应弹珠的 JSON；第 90 ~ 94 行基于弹珠 JSON 中的数据，构建“owner~name”组合键，并利用此键删除当前弹珠所有人与弹珠的映射关系；第 96 行和第 97 行从 Fabric 账本中删除此弹珠。

（4）实现智能合约的 main 函数

```
112     public static void main(String[] args) throws Exception {
113         log.info("starting");
114         new MarbleExample().start(args);
115     }
```

智能合约的 main 函数其实现都是类似的，构建智能合约对象，并调用其 start 接口，启动智能合约服务。

6.8.4 开发和提交代码

（1）环境要求

推荐在 Linux（如 CentOS 7.1+ / Ubuntu 14.04+）或 MacOS 环境中开发代码，并安装如下工具。

- git：用来获取代码。
- golang 1.6+：安装成功后需要配置 $GOPATH 等环境变量。
- Docker 1.12+：用来支持容器环境，注意 MacOS 下要用 Docker for Mac。

（2）获取代码

首先注册 Linux foundation ID，并登录 https://gerrit.hyperledger.org/，添加个人 ssh pub-key。找到 Fabric 项目，采用 Clone with commit-msg hook 的方式来获取代码。

执行如下命令获取代码，并放到 $GOPATH/src/github.com/hyperledger/ 路径下，将其中的 LF_ID 替换为你的 Linux Foundation ID。

```
$ mkdir $GOPATH/src/github.com/hyperledger/
```

```
$ cd $GOPATH/src/github.com/hyperledger/
$ git clone ssh:// LF_ID@gerrit.hyperledger.org:29418/fabric && scp -p -P 29418
LF_ID@g errit.hyperledger.org:hooks/commit-msg fabric/.git/hooks/
```

如果没有添加个人 ssh pubkey，则可以通过 HTTPS 方式 clone，需要输入用户名和密码信息：

```
git clone http:// LF_ID@gerrit.hyperledger.org/r/fabric && (cd fabric && curl -kLo
`git rev-parse --git-dir`/hooks/commit-msg http:// LF_ID@gerrit.hyperledger.org/r/tools/hoo
ks/commit-msg; chmod +x `git rev-parse --git-dir`/hooks/commit-msg)
```

（3）编译和测试

大部分编译和安装过程都可以利用 makefile 来执行，包括如下的常见操作。

安装 go tools，执行如下命令：

```
$ make gotools
```

（4）语法格式检查

执行如下命令检查语法格式：

```
$ make linter
```

（5）编译 peer

执行如下命令编译 peer：

```
$ make peer
```

此时，就会会自动编译生成本地 peer 可执行文件。

有时候会因为获取的安装包不稳定而报错，需要执行 make clean，然后再次执行。

（6）生成 Docker 镜像

生成 Docker 镜像时，需要执行如下命令：

```
$ make images
```

（7）执行所有的检查和测试

执行如下命令，以进行检查和测试：

```
$ make checks
```

单元测试的命令如下：

```
$ make unit-test
```

如果要运行某个特定的单元测试，则可以通过类似如下的格式来实现：

```
$ go test -v -run=TestGetFoo
```

进行 BDD 测试时，需先生成本地 Docker 镜像。可通过如下命令来实现：

```
$ make behave
```

（8）提交代码

使用 Linux foundation ID 登录 jira.hyperledger.org，查看有没有未分配的任务 / 问题，如果想要处理某个任务，可以添加自己为 assignee，如对 FAB-XXX 任务。

本地创建新的分支 FAB-XXX 的命令如下：

```
$ git checkout -b FAB-XXX
```

```
$ git commit -a -s
```

实现任务代码，完成后，进行语法格式检查和测试等，确保所有检查和测试都能通过。提交代码到本地仓库。

此时，会打开一个窗口需要填写 commit 信息，格式一般要求为：

```
Simple words to describe main change This fixes #FAB-XXX.
A more detailed description can be here, with several paragraphs and sentences...
```

之后使用 git review 命令推送到远端仓库：

```
$ git review
```

提交成功后，可以打开 gerrit.hyperledger.org/r/，查看自己最新提交的补丁集信息，添加几位审阅者。之后就是等待开发者团队的审阅结果，如果结果通过，则会被项目的维护人员合并到主分支。否则还需要针对大家提出的建议做进一步的修正。

修正过程与提交代码的过程类似，唯一不同是，提交的时候使用：

```
$ git commit -a --amend
```

表示这个提交是对旧提交的一次修订。

相关术语

- Auditability（审计性）：在一定权限和许可下，可以对链上的交易进行审计和检查。
- block（区块）：代表一批得到确认的交易信息的整体，准备被共识加入到区块链中。
- blockchain（区块链）：由多个区块链接而成的链表结构，除了首个区块之外，每个区块都包括前继区块内容的 Hash 值。
- chaincode（智能合约）：作为交易的一部分保存在总账上的应用级的代码（如智能合约），其执行结果会改变区块的世界状态，支持多种开发语言，支持 golang、Node.JS、Java 等。
- Committer(提交节点)：Fabric 网络中的一种节点角色，负责对排序后的交易进行检查，

选择合法的交易执行并写入持久化存储。

- Confidentiality（保密）：只有交易相关方才可以看到交易的内容，其他人若未经授权则无法看到。
- Endorser（背书节点）：Fabric 网络中的一种节点角色，负责检验某个交易是否合法，是否愿意为之背书、签名，生成读写操作集合。
- Ledger（账本）：包括区块链结构（带有所有的交易信息）和当前的世界观（world state）。
- MSP（Member Service Provider，成员服务提供者）：成员服务的抽象访问接口，实现对不同成员服务的可拔插支持。
- Orderer（排序节点）：Fabric 网络中的共识服务角色，负责排序交易，提供全局确认的顺序，例如，使用 Kafka 或 PBFT。
- Permissioned Ledger（许可制账本）：网络中所有节点都必须是经过许可的，非许可过的节点则无法加入网络。
- Privacy（隐私保护）：交易员可以隐藏自己在网络上的身份，其他成员在无特殊权限的情况下，只能对交易进行验证，而无法获知交易员的身份信息；只有交易双方才可以看到交易内容，其他人若未经授权则无法看到，且无法回溯到交易方。
- Transaction（交易）：执行账本上的某个函数调用。具体函数在 chaincode 中实现。
- Transactor（交易者）：发起交易调用的客户端。
- World State（世界观）：是一个 key/value 数据库，chaincode 用它来存储交易相关的状态交易——区块链上执行功能的一个请求。功能是使用智能合约来实现的。

Appendix A 附录 A

国内区块链联盟介绍

A.1 中国区块链研究联盟（CBRA）

“中国区块链研究联盟”，于2016年1月5日在北京正式成立，英文名称为“China Blockchain Research Alliance”（以下简称“联盟”或CBRA），由全球共享金融100人论坛联合论坛理事单位共同发起。

A.1.1 成立背景

2016年1月20日，中国人民银行数字货币研讨会在北京召开。来自中国人民银行、花旗银行和德勤公司的数字货币研究专家分别就数字货币发行的总体框架、货币演进中的国家数字货币、国家发行的加密货币等专题进行了研讨和交流。中国人民银行行长周小川出席会议，中国人民银行副行长范一飞主持会议。会议指出，数字货币的发展正在对中央银行的货币发行和货币政策带来新的机遇和挑战。中国人民银行对此高度重视，从2014年起就成立了专门的研究团队，并于2015年年初进一步充实力量，对数字货币发行和业务运行框架、数字货币的关键技术、数字货币发行的流通环境、数字货币面临的法律问题、数字货币对经济金融体系的影响、法定数字货币与私人发行数字货币的关系、国际上数字货币的发行经验等进行了深入研究，已取得阶段性成果。

CBRA是专业的区块链学术研究平台，研究成员由国内外学界、实业界具有较强学术功底和社会影响力的专家担任，立足于打造区块链技术的研究平台与交流平台；打造政策沟通平台，厘清区块链技术在现有监管模式与货币政策操作中的定位；打造区块链技术的市场应用平台，推动具体应用规则的规范化、标准化，进行项目落地与路演，形成区块链研究领域

具有高学术品味和国际影响力的中国特色新型智库，成为全球共享金融 100 人论坛中具有基础作用的重要组成部分。

A.1.2　主要工作

CBRA 的主要工作包含如下几个方面

1）组织“全球区块链研究联盟峰会”，聚集全球专业人士，共同探讨技术应用。

2）组织区块链联盟沙龙，选择不同主题进行深入探讨，对业界适度开放，以扩大联盟影响力。

3）构建数个研究小组，每年出一个报告，最后汇总出版年度《中国区块链发展报告》。

4）打造区块链创新项目应用机制。推动项目落地、进行项目路演、构建区块链相关产业合作生态体系。

5）编辑内报《中国区块链研究联盟动态资讯》，每两月一期。每期内容包含联盟成员的原创文章，以及最新的资料信息收集等。

6）设立客座研究员机制，包括资深研究员、高级研究员、研究员三类，人员聘用由 GSF100 理事委员会、学术委员会讨论通过。

A.1.3　组织架构

主办单位：全球共享金融 100 人论坛

发起单位：中国万向控股有限公司、厦门国际金融技术有限公司、中国保险资产管理业协会、营口银行股份有限公司、北京太一云科技有限公司

学术委员会：

中国区块链研究联盟主任

杨　涛　全球共享金融 100 人论坛学术委员会副主任
　　中国社会科学院金融研究所所长助理

中国区块链研究联盟副主任

肖　风　全球共享金融 100 人论坛学术委员会副理事长
　　中国万向控股有限公司副董事长兼执行董事

曹　彤　全球共享金融 100 人论坛副理事长
　　厦门国际金融技术有限公司董事长

中国区块链研究联盟高级研究员

高林挥　　乐视金融高级运营总监、乐视金融区块链实验室创始人

申屠青春　银链科技创始人

曹　锋　深圳瀚德创客投资有限公司 CIO

王立仁　北京人民汇金科技有限公司总裁

龚　鸣　区块链铅笔创始人

蒋　海　布比创始人
金　巍　中国文化金融 50 人论坛秘书长
　　中国传媒大学文化经济研究所研究员

A.1.4 举办会议

1）区块链研究联盟成立会议，2016 年 1 月 5 日
2）区块链·黄埔一期培训，2016 年 2 月 26 至 2016 年 2 月 28 日
3）共享金融与区块链，深圳峰会 2016 年 3 月 12 日
4）区块链沙龙三：解密数字货币，2016 年 5 月 7 日
5）中国大数据产业峰会·中国区块链金融分论坛贵阳，2016 年 5 月 26 日
6）区块链·黄埔二期，2016 年 6 月 2 日至 2016 年 6 月 4 日
7）中国区块链产业大会揭秘六大行业应用，2016 年 8 月 21 日

A.2 金融区块链合作联盟（深圳)(金链盟，FISCO）

随着数字经济的发展及数字资产自由流转需求的驱动，全球范围跨行业跨区域的新型信任机制的逐步构建，区块链技术正在成为关注焦点。

广义而言，区块链技术是一种利用块链式数据结构来验证与存储数据、利用分布式节点与共识算法来生成和更新数据、利用密码学的方式保证数据传输和访问的安全、利用由自动化脚本代码组成的智能合约来编程和操作数据的分布式基础架构与计算范式，按部署模式可以划分为公有链、私有链、联盟链三种形式。

发展到今天，区块链已经融汇吸收了分布式架构、块链式数据验证与存储、点对点网络协议、加密算法、共识算法、身份认证、智能合约、云计算等多类技术，并在某些领域与大数据、物联网、人工智能等形成交集与合力，现已成为一种整体技术解决方案的总称。

目前国外金融巨头纷纷布局区块链领域，区块链创业公司 R3CEV 发起的 R3 区块链联盟，主要致力于为银行提供基于区块链技术的解决方案，并推出了适用于金融机构的分布式账本 Corda；Linux 基金会发起的推进区块链数字技术和交易验证的开源项目超级账本（HyperLedger），旨在让成员共同合作，共建开放平台。

为研发适合我国国情的区块链底层技术，促进区块链在金融行业的应用及落地，凝聚力量并提升我国企业在区块链技术领域的话语权，金融区块链合作联盟（深圳)(简称“金链盟”）应运而生。

A.2.1 金链盟概况

在深圳市金融办、市科协及各监管部门的关怀下，经过精心筹划，2016 年 5 月 31 日，深金信会、银链科技、深证通、微众银行、安信证券、京东金融、博时基金、重庆股转中

心、第一创业证券、富德保险控股、国信证券、恒生电子、南方基金、齐鲁股交中心、金证股份、赢时胜、致远速联、四方精创、武交中心、招商证券、招银网络、中股集团、中证信用等 20 余家金融机构和金融科技企业，共同发起并成立金融区块链合作联盟（深圳）。金链盟还吸纳了华为、腾讯、万达网络科技、广发银行、江苏银行、包商银行、长沙银行、洛阳银行、上饶银行、华瑞银行、华安财险、前海股转、前海人寿、山东城商行合作联盟等知名机构加入。目前，金链盟成员包括银行、基金、证券、保险、地方股权交易所、科技公司等六大类行业的 60 余家机构。

金链盟是开放式组织，自愿遵守章程的金融机构及向金融机构提供科技服务的企业等均可申请加入。

金链盟旨在整合及协调金融区块链技术研究资源，形成金融区块链技术研究和应用研究的合力与协调机制，提高成员单位在区块链技术领域的研发能力，探索、研发、实现适用于金融机构的金融联盟区块链，以及在此基础之上的应用场景。

金链盟的组织架构如图 A-1 所示。

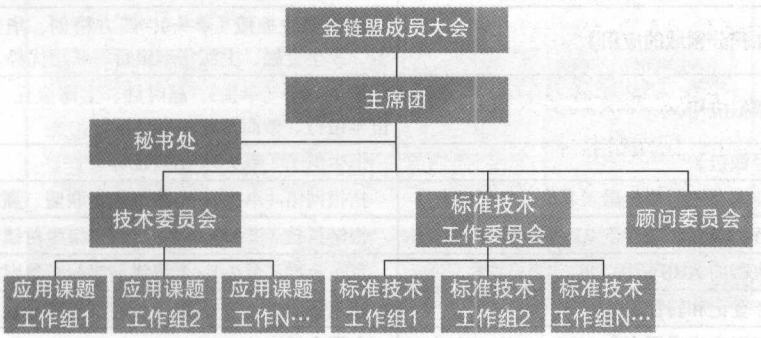


图 A-1　金链盟组织架构

联盟成员大会是联盟的最高权力机构；联盟设主席团，主席团是成员大会的常设机构，在成员大会闭会期间领导本联盟开展日常工作，对成员大会负责；标准技术工作委员会负责金链盟团体标准的立项、制定、审定和发布等工作；经主席团批准，外部专家可以顾问或观察员的身份参与标准研讨。

目前，金链盟已建立了联盟网站（www.fisco.com.cn），并开设了微信公众号“FISCO 金链盟”，作为联盟对外宣传的窗口，发布联盟动态，便于外界及时了解金链盟在区块链技术和标准研究方面的进展。

A.2.2　金链盟在区块链技术方面的研究课题及成果

在区块链课题研究方面，金链盟在云服务、数字资产、信用、股权、理财产品发行及交易、积分、保险、票据等领域逐渐从理论探讨走向实践应用。金链盟内部已经开展讨论并审定了较为细致的 13 个应用课题研究发展方向（见表 A-1）。

表 A-1 金链盟课题任务表

序号	项目名称	承担单位
1	《基于区块链的场外股权交易市场平台》	深证通（牵头）、前海股交、中股集团、重庆股转、齐鲁股交、武汉股交、致远速联、银链、金证股份、招商证券、鑫苑置业、粤股交
2	《区块链底层技术平台》	银链（牵头）、微众银行、四方精创、金证股份、南方基金、山东城商联盟、赢时胜、富德保险控股、招商证券、华安保险、京东金融、深证通、泰康人寿、厚生金融、粤财汇、博彦科技、鑫苑置业、前海人寿、光大证券
3	《区块链云服务》	微众银行（牵头）、腾讯、银链、赢时胜、四方精创、越秀金控
4	《区块链理财产品一二级市场》	微众银行（牵头）、安信证券、四方精创、博时基金、山东城商联盟、第一创业、赢时胜、京东金融、重庆股转、银链科技、厚生金融、上海金丘、万达网络科技、粤股交
5	《区块链信用服务》	中证信用（牵头）、国信证券、金证股份、南方基金、恒生电子、银链、万达网络科技、粤财汇、鑫苑置业、越秀金控、粤股交
6	《区块链在积分领域的应用》	富德保险控股（牵头）、四方精创、华安保险、前海人寿、厚生金融、上海华瑞银行、联动优势、鑫苑置业
7	《区块链票据应用》	恒生电子（牵头）、赢时胜、上海金丘、上海华瑞银行、恒丰银行、徽商银行、泰隆银行
8	《以太坊轻钱包》	恒生电子（牵头）、招商证券
9	《区块链技术在银行业金融工具交易中的应用》	招银网络（牵头）、山东城商行联盟、赢时胜、博彦科技
10	《商业银行抵押品区块链》	银链科技（牵头）、四方精创、宝生村镇银行
11	《基于区块链的 ABS 云服务》	京东金融（牵头）、上海华瑞银行、粤财汇、泰隆银行
12	《数字资产登记和转让》	京东金融（牵头）、重庆股转、上海金丘、光大证券
13	《海星数字资产流通平台》	上海金丘

其中部分课题项目已经落地或推出 DEMO 版产品，如图 A-2 所示。

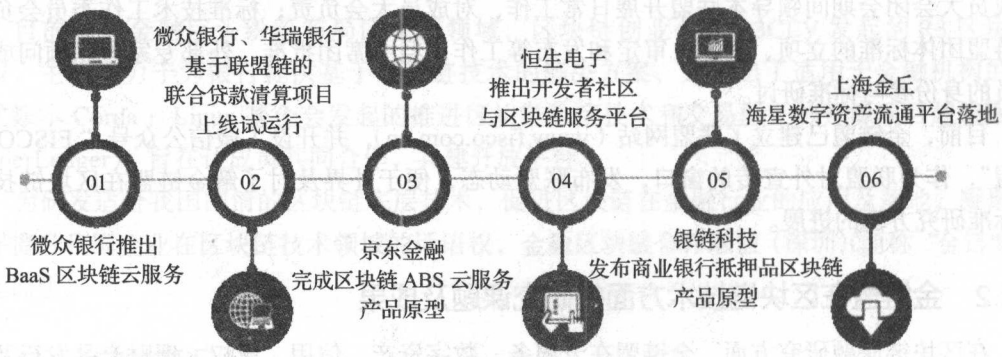


图 A-2 金链盟成员在区块链领域的技术成果

以下对部分金链盟课题做简要介绍。

1. 深证通《基于区块链的场外股权交易平台》

在股权交易方面，面对区域股权分散、中介机构执业信息无法共享及征信标准不统一等行业痛点，深证通在 2016 年 7 月提出了基于区块链技术的解决方案，并联合金链盟相关成员，发起《基于区块链的场外股权交易平台》课题研究。

在解决方案上，课题组提出通过建立一个征信链，为股交中心、中介机构、企业及监管机构提供一个征信信息上传及查询平台；在技术实现上，该课题组基于深证通金融云成熟的基础设施，打造深证通金融区块链平台，支持中介机构征信等业务应用，从而推动区域股权市场中介机构征信标准的制定，促进市场业务的发展，打通区域股权市场互连互通的信息共享路径。

中介机构征信链技术实现架构如图 A-3 所示。

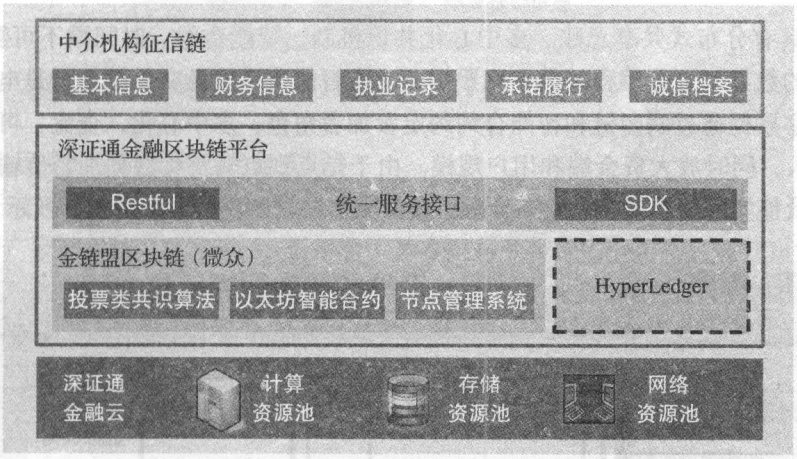


图 A-3　中介机构征信链技术实现架构（资料来源：深证通）

2. 微众银行推出区块链云服务 BaaS

建立和维护区块链与分布式基础设施，需要大量的手动开发工作及强大的后端云计算能力，因此，微众银行联合腾讯打造了“区块链云服务 BaaS”。其宗旨是为金链盟成员的技术验证、业务模型验证及开发研究等提供底层技术支持。目前公测版本已部署就绪可接入试用，这是国内第一个面向金融业的云服务，也是国内区块链领域中第一个发布的金融联盟链产品。

对应用场景的开发者而言，既能够共享区块链的底层设施（包括共享云服务相关的技术、软件和代码，不需要每个开发成员重复投入），又能使用友好、简单、跨平台的应用开发 API 与图形化管理平台及区块链浏览器等，加速开发流程，改善区块链产品的创建和管理体验。通过“区块链云服务 BaaS”，接入的机构可以轻松自建联盟链，使用高效的 PBFT 交易共识机制，并控制审批区块链上的节点身份，实现轻量级、易管理、快速开发应用场景的目的。区块链云服务 BaaS 架构如图 A-4 所示。

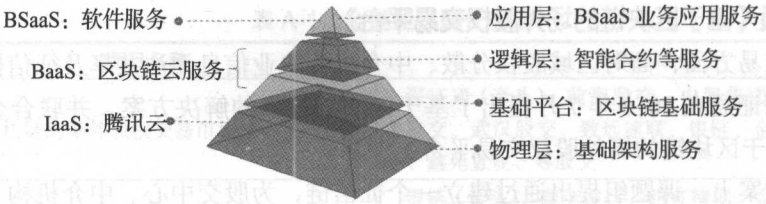


图 A-4 区块链云服务 BaaS 架构（资料来源：微众银行）

3. 京东金融关于数字票据的研究与概念证明

现行的票据在实践中存在着诸如贸易背景造假、一票多卖、背书不连续、审核困难成本高等问题。如何解决这些问题，是金融界的难点之一。区块链技术的出现，仿佛让人们看到了曙光。

区块链具有分布式共享总账、多中心化共识机制、智能合约、时间戳不可篡改等特征。基于区块链的数字票据是编程的数字化票据，支持智能化风控及交易结算，是电子票据的有益补充。数字票据通过联盟链和智能合约约定参加者角色，多中心化，全流程可审计；通过 API 一点接入，同时放大资金端和用户规模。由于是点到点非对称数据加密传输，且数据不可篡改，因此能实现隐私保护。京东金融区块链数字票据解决方案如图 A-5 所示

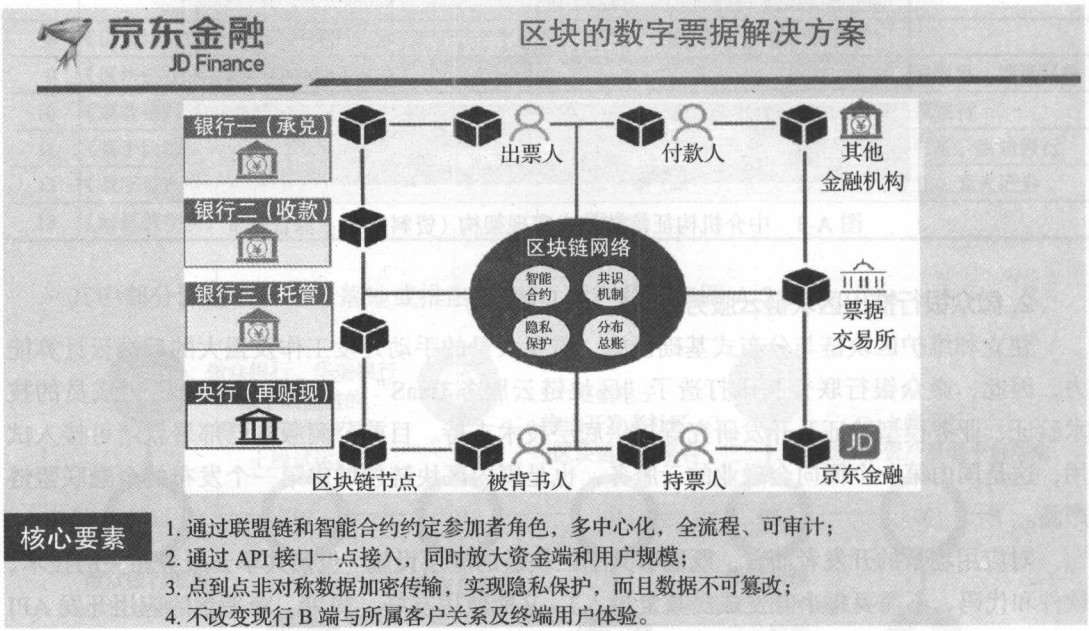


图 A-5 京东金融区块链数字票据解决方案（资料来源：京东金融）

4. 上海华瑞银行基于区块链的个人跨境汇款业务

华瑞银行认为在应用前景上，未来线上的身份验证、交易确认是区块链应用的重点，特

别是在跨境频繁的交易与资金清算领域，区块链技术会得到广泛的应用。基于区块链的跨境支付能有效改变传统模式效率低、速度慢、手续费贵的现状，如通过 Ripple 提供的区块链分布式账本技术，与代理行之间在区块链上直接进行资金的结算，既可实时到账，最大又可节省 75% 的手续费。华瑞银行跨境支付基础设施架构如图 A-6 所示。

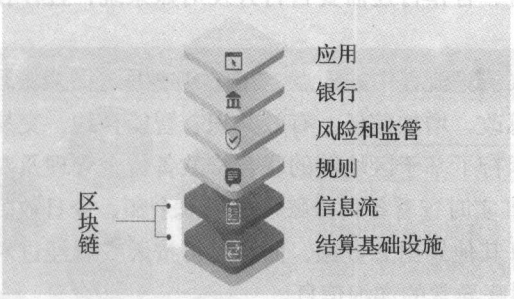


图 A-6　华瑞银行跨境支付基础设施架构（资料来源：华瑞银行）

5. 上海金丘《海星链规划与金融实践》

上海金丘通过将区块链与传统金融技术糅合，构建自主的海星区块链技术平台、BaaS 云平台、数字资产登记系统、RobotABS 云平台，为金融行业带来安全可信、高效稳定的区块链金融解决方案。目前各平台框架已搭建完成，后续将会与多家金融机构、企业服务商和技术提供方共同开展合作，发掘更多的金融业务场景。金丘区块链平台连接器方案如图 A-7 所示。

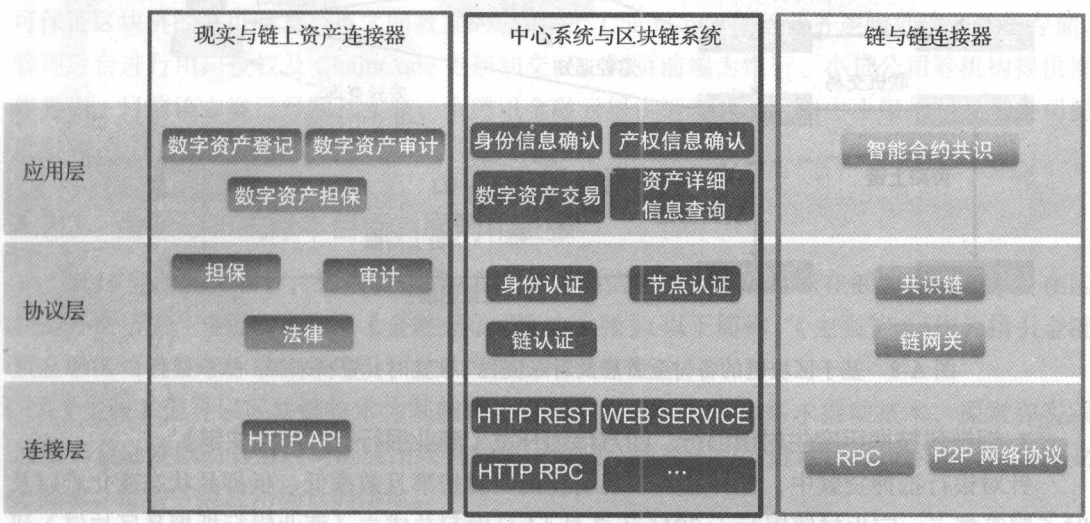


图 A-7　金丘区块链平台连接器方案（资料来源：上海金丘）

6. 微众银行与华瑞银行携手开发基于区块链的联合贷款备付金管理及对账平台

2015 年 5 月，微众银行面向微信用户和手机 QQ 用户推出纯线上个人小额信用循环消费

贷款产品——“微粒贷”，并得到了快速的发展，但同时在运营效率、管理服务能力方面也遇到了急需改进的问题。在传统的联合贷款对账清算流程中，合作行无法实时（必须在批量对账后的 T+1 日）了解到引发备付金账户变动的贷款借还交易明细信息及对账信息，无法及时了解到账务是否不平，这不便于合作行及时配置备付金账户、及时管理流动性头寸、及时处理账务异常情况等。同时，合作行还需要自行开发对账系统，且存在缺乏统一全面的信息视图等问题。

针对在联合贷款对账清算流程中的痛点，2016 年 9 月，微众银行与华瑞银行携手，充分利用区块链技术的不可篡改、增加信任、有效共识、智能合约、交易快速确认、可追溯等特性，共同开发并上线试运行了基于区块链的联合贷款备付金管理及对账平台。合作行通过此系统可以在 Web 页面上准实时查看备付金账户的当前余额、今日放款总金额、今日还款总金额、今日其他划入和今日其他划出总金额，而且还可按天查看经过对账后的每日汇总信息及流水对账结果，可查看对账异常的详细信息。

该备付金管理及对账平台的最大优势在于，让机构与机构间的互信关系转变为共同对技术的信任关系，把此前需要 T+1 的对账周期缩短到准实时。此外，通过智能合约的使用还可实现实时流水对账、准实时发现对账差异、实时计算合作行备付金账户当日发生额与当日余额等功能。基于区块链的备付金管理及对账流程如图 A-8 所示。

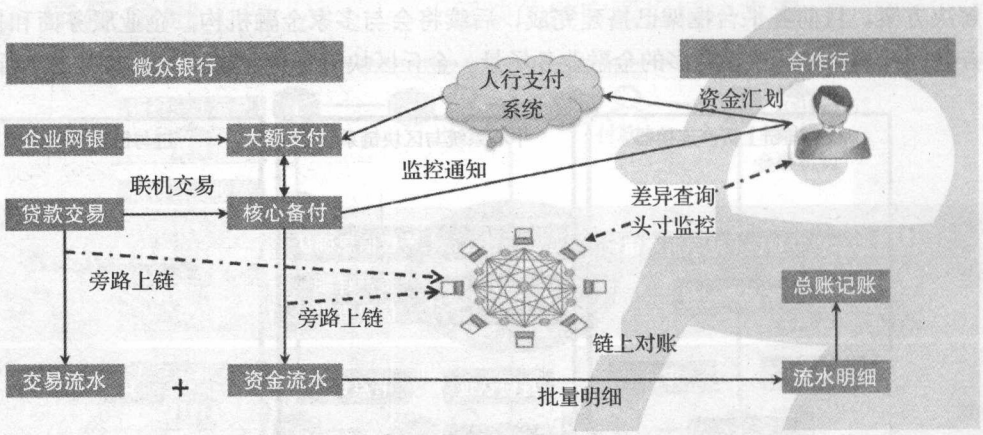


图 A-8 基于区块链的备付金管理及对账流程（准实时）（资料来源：微众银行）

7. 银链科技携手宝生村镇银行、四方精创开发《商业银行抵押品区块链》

针对银行抵押贷款中，抵押品人工现场查询费时费事且效率低、抵押品状态变化难以及时了解等痛点，宝生村镇银行于 2016 年 7 月 14 日发起并成立《商业银行抵押品区块链》研究课题，为解决上述问题提出基于区块链技术的解决方案。银链科技、四方精创等联盟成员负责该课题的技术实施。课题原型开发于 2016 年 9 月 15 日顺利完成，并完全实现了最初设计的功能。

该方案中，抵押涉及的所有单位及个人，如银行、小贷公司、车管所、国土局、房管所、评估机构等多方组成一个联盟链，保证同一业务的相关方全部参与。并且统一区块链接口，数据实时共享，查封、贬值等状态可以随时得知，同时各方机构在审核时可以方便快捷地查询该抵押品有无抵押，而无须去现场。基于区块链的商业银行抵押品登记解决方案如图 A-9 所示。

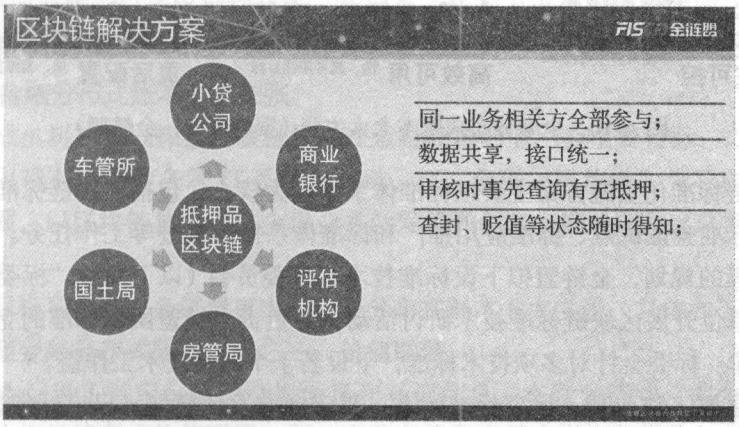


图 A-9　基于区块链的商业银行抵押品登记（资料来源：银链科技）

在具体技术实施上，抵押品区块链是基于 IBM 的 Fabric 区块链平台而开发的，包括 IBM Fabric 区块链平台、智能合约 ChainCode、管理后台和网页前端四大部分。其中 Fabric 可保证区块链的许可制及节点之间数据的一致性，ChainCode 用于业务逻辑处理和数据存储，管理后台进行用户授权及 ChainCode 更新和交互，网页前端为银行、小贷公司等机构提供操作界面。目前该方案已经顺利实施，由腾讯金融云提供运营环境，向中小银行和小贷公司提供服务。

A.2.3　金融分布式账本目标、原则和主张

2016 年 11 月 26 日，第八届中国（深圳）金融信息服务发展论坛在深圳福田举办。在区块链分论坛上，金链盟发布了《金融分布式账本主张》（以下简称“《主张》”），作为指引金链盟金融区块链技术标准体系建设和技术发展的纲领性文件。

《主张》倡导以区块链技术为基础建设金融分布式账本，以需求推动技术，探索和实践适用于金融领域的联盟区块链应用架构和使用原则，研究如何改进区块链技术以满足金融业务要求，推动各类应用场景的落地。

《主张》提出**合法合规、可追溯、安全、隐私保护、业务导向**的五大原则，和**价值联盟、自主可控、安全可信、高效可用、业务可行、灵活配置、智能监管**的七大主张，（如图 A-10）其目标是让成员机构形成对金融区块链技术的统一认识，并在此之上，规范业务与应用的标准、开发与部署环境的标准、监管审计的标准等。

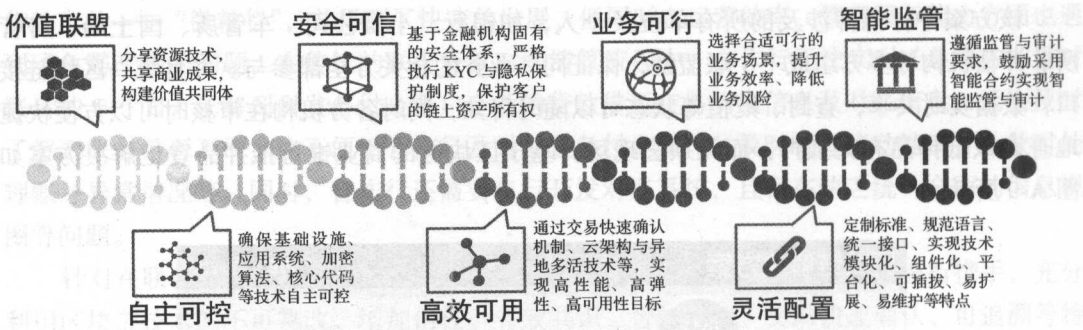


图 A-10 金融分布式账本七大主张（资料来源：金链盟）

金融区块链标准化实施方案主要以标准体系建设为核心，包括区块链标准体系研究、重点标准研制、标准验证试点、标准应用推广和标准体系持续改进等工作任务。为落实区块链技术标准化进程的规划，金链盟拟下设标准技术工作委员会（以下简称“标委会”），旨在组织金链盟成员单位开展区块链标准技术研讨活动，并负责金链盟团体标准的立项、制定、审定和发布等工作。标委会针对多项技术标准，下设若干个标准技术工作组。

标委会的主要工作包括：研究标准体系，制定标准发展规划，提出标准技术工作项目，通过标准工作计划，审查标准工作成果并征询意见，与国内外标准组织合作研究，衔接国内外标准，决定标准化工作组的设立、调整或撤销，组织标准相关的学术活动等。

标委会将以金融分布式账本七大主张为纲领，针对金融联盟链场景的应用需求，制定金融联盟链技术的参考架构与功能架构，包括但不限于成员管理、数字身份认证、数字资产登记管理、共识算法、智能合约、加密算法、隐私保护、监管审计等基本原则，从而为金链盟成员单位的区块链技术开发与应用场景实施提供参考，指引基于联盟链的金融业务开展，保障相关业务与活动安全，形成统一的技术规范以促进业务的互连互通。

A.2.4 金融分布式账本主张

1. 目标和原则

随着数字经济的发展及数字资产自由流转需求的驱动，全球范围跨行业跨区域的新型信任机制的逐步构建，国内外金融巨头纷纷布局区块链，金融区块链合作联盟（深圳）（以下简称“金链盟”）也由此应运而生。金链盟倡导以区块链技术为基础，建设金融分布式账本，以需求推动技术，探索和实践适用于金融领域的联盟区块链应用架构和使用原则，研究如何改进区块链技术以满足金融业务要求，推动各类应用场景的落地。为加快以联盟链为基础的金融分布式账本技术的成熟应用，提出以下五大基本原则。

1）合法合规原则：分布式账本建设各项活动中，金链盟各成员机构应遵守国家相关法律法规和金融监管的要求，并从技术上支持各种监管和审计的需要。

2）可追溯原则：分布式账本上进行的业务与活动都应有记录，以满足可追溯和审计的

要求。

3）安全原则：分布式账本及其上的业务应用应采取各种必要的安全手段，保障链上资产和交易等信息的安全，以防范外部和内部的攻击。

4）隐私保护原则：基于分布式账本上的业务应用应通过加密、签名、权限控制等各种安控手段，保障链上用户隐私的安全，防止泄露用户隐私。

5）业务导向原则：以需求推动技术，在分布式账本上进行业务开发时，应优先考虑业务场景，保障业务的可靠性、时效性和稳定性。

2. 金链盟金融分布式账本七大主张

金链盟各成员机构是以探索区块链技术在金融领域的应用为主要目标，以国家各项法律法规为指导，以现有金融业务和IT系统为基础，以“金融＋区块链”为导向，建设金融分布式账本生态圈，以区块链技术促进金融业务的发展和创新，推动信息互联网向价值互联网的转变。

基于上述目标和原则，金链盟提出了金融分布式账本七大主张，具体如下

（1）金链盟金融分布式账本主张之一：**价值联盟**

分布式账本记录了特定价值的交换信息和登记信息，在联盟链中，多个业务参与方基于一个特定的价值目标形成价值联盟，每一个分布式账本的实现，都是支持特定价值联盟的业务开展。而对于每个价值联盟，其行业属性、业务模式、共同价值目标的差异，又造就了其分布式账本的差异化，特定的业务拥有特定的分布式账本。在金融领域中，不同的价值资产，不同的业务模式，采用不同的分布式账本支持相应价值联盟的业务目标。

（2）金链盟金融分布式账本主张之二：**自主可控**

基于对金融机构所开展业务及系统的合规、安全和稳定等方面的要求，现有的国际区块链与分布式账本技术，或难以适应我国的金融环境，或难以达到大规模商用的技术要求，因此金链盟主张坚持以自主可控为原则，根据实际国情定制金融分布式账本，逐步构建自主生态系统，最终实现整体架构、分布式账本网络、关键基础设施、应用系统、加密算法、核心代码等的自主可控，并以此为基础，鼓励各类机构对传统业务模式与业务流程进行梳理，寻求以分布式账本技术进行流程再造和业务创新。

（3）金链盟金融分布式账本主张之三：**安全可信**

金融业务具备严格牌照资质、强隐私保护、网络与资金高安全要求、反洗钱、反欺诈等特性，需要从以下几个方面构筑金融分布式账本的安全体系，保障业务及活动的安全可信。

一是基于金融机构固有的安全防护体系之上建设分布式账本的基础设施，并且结合区块链技术自身的加密、签名等安全机制，确保交易安全、防篡改和防抵赖。

二是根据金融机构KYC要求，对接入金融分布式账本的机构和个人进行实名身份认证，实施准入控制，满足反洗钱、反欺诈等方面的要求。

三是实现严格的客户隐私保护，按“最小需要”原则，在机构间传递和使用满足正常业务需要的最小数据集合，并通过权限控制、数据隔离、数据加密等手段进行访问控制，避免

客户隐私泄露。

四是通过建立用户与链上资产的所有权关系，实现金融资产安全保护。例如：允许用户在账户凭证丢失 / 泄露的情况下，通过挂失、冻结等手段保护资产免受损失，通过变更所有权关系避免资产丢失。

（4）金链盟金融分布式账本主张之四：**高效可用**

结合区块链技术的特点，可以通过以下措施来实现高性能、高弹性、高可用性的目标。

一是金融分布式账本建立在联盟链技术的基础之上，通过可控的共识机制来优化共识算法，相比公有链技术，能够缩短共识时间，通过剔除不必要的算力证明等因素，将资源集中在交易处理环节，从而改善交易性能。

二是将节点部署在可灵活伸缩的分布式架构之上，使金融分布式账本具备高效快速扩容的能力，实现架构整体容量和单位处理性能的快速提升。

三是根据金融监管要求，按照“同城多活加异地灾备”的标准建设联盟链系统，对节点间通信链路进行容灾设计，实现系统、数据和链路的冗余备份，保证系统部署架构的可靠性和可用性；同时，基于联盟链一致的点对点协议所构成的软件异构环境，确保了即便某个节点中某个版本的软件出现问题，联盟链的整体网络也不会受到影响，从而进一步保证了分布式账本的高可用性。

（5）金链盟金融分布式账本主张之五：**业务可行**

区块链技术具有分布式、可追溯、不可篡改、增加信任、集体维护、可靠数据库等优点，金融业务可以根据自己的场景痛点来应用区块链技术，在综合考虑区块链技术的特定特性与金融监管环境下，把分布式账本作为一个技术组件，优化现有业务流程，并通过更多业务场景的应用来提高分布式账本技术的成熟度，以及完善分布式账本技术在金融行业的业务支持能力。例如：利用分布式账本的分布式、不可篡改特性优先改善非中心化和多中心化的金融机构间的清结算业务，提高时效，降低资金成本。

（6）金链盟金融分布式账本主张之六：**灵活配置**

由于成员机构间技术选择和开发模式的差异，需要有统一的技术平台和标准化的技术组件，来降低技术难度，节省成本，提高成员在区块链技术领域的研发能力，快速构建企业级应用，推动区块链技术在金融行业的发展。为达到这一目标，可通过规范开发语言、制定统一接口协议、定制标准 API 和标准组件模版等途径，实现技术模块化、组件化、平台化。例如：在智能合约的编写语言上优先选择金融业常用语言，以利于业务快速移植；通过接口协议标准化和 API 标准化实现组件可插拔；通过将具有共性的金融服务固化为标准组件模版或基础服务模块，实现业务的快速落地等。

（7）金链盟金融分布式账本主张之七：**智能监管**

依据各种法规框架、条文、行业政策、业务规范和技术要求，金融分布式账本的监管与治理规则通常分为两大层面：一是技术系统之内可自动化实现的治理规则，由软件、协议、算法、智能合约、配套设施等技术要素构成；二是需要人参与监督管理的无法用技术自动实

现的规则，由组织机构或管理人员进行监管治理。金链盟主张两者兼顾，确保联盟链上的所有活动都符合监管要求，鼓励充分利用智能合约等技术有效支持审计监管操作。从全局的角度，建议通过事前准入控制、事中权限控制和全局交易控制、事后审计等手段保证交易记录防篡改、可追溯与可审计，实现监管目标。

3. 总结和展望

总而言之，金链盟作为在金融领域探索和应用区块链技术的先行者，将根据金融分布式账本的五大原则和七大主张，通过金融分布式账本的不断迭代应用，理论与实践相结合，以需求推动技术进步，依托技术发展探索更多应用场景，推动上述原则和主张在金融实践活动中得到有效的体现，助力金融领域的技术创新和可持续发展。

A.3 中国分布式总账基础协议联盟（ChinaLedger）

A.3.1 联盟成立

2016 年 4 月 19 日晚，中国分布式总账基础协议联盟（以下简称 ChinaLedger 联盟）在北京正式宣布成立，这是亚洲首个专注于分布式账本及其衍生技术研究的技术性联盟。上海证券交易所前总工程师白硕先生担任该联盟的技术委员会主任。该次大会也是 ChinaLedger 联盟的第一次发起人大会。

ChinaLedger 联盟的创始成员包括中证机构间报价系统股份有限公司、中钞信用卡产业发展有限公司北京智能卡技术研究院、浙江股权交易中心、深圳招银前海金融资产交易中心、厦门国际金融资产交易中心、大连飞创信息技术有限公司、通联支付网络服务股份有限公司、上海矩真金融信息服务有限公司、深圳瀚德创客金融投资有限公司、乐视金融、万向区块链实验室等十余家单位组成。万向区块链实验室是 ChinaLedger 联盟的秘书处。

联盟的成立，是各个共同发起方的共识。在区块链技术向各个领域特别是金融领域迅速渗透的新形势下，各国的金融机构、资本市场、信息技术领域都面临着巨大的挑战和机遇。利用区块链技术打造价值互连互通的基础设施，已经被全球很多国家提到了战略的高度。由全球性的金融机构组成的跨境区块链联盟已经在积极行动并推出了阶段性成果。

ChinaLedger 的成员认为，各机构要深入研究、积极跟进区块链技术在金融科技领域带来的新变革、新动向，积极争取我国在国际区块链领域的话语权，防止在新一轮金融科技革命中被边缘化；另一方面，也需要在区块链技术落地时充分考虑我国金融监管和风险管控制度安排的特殊性，妥善应对新技术在实施过程中对金融体系可能带来的影响。

A.3.2 联盟进展

在 ChinaLedger 成立之后的一段时间后，联盟又吸收了一部分观察员，观察员所在的单位大部分是金融机构，也有部分是研究机构。联盟也聘请了 4 位海外顾问，其中包括加拿大

多伦多交易所首席数据官、Bitcoin 的核心开发者、R3 联盟的市场主管，以及以太坊的创始人。

ChinaLedger 成员共同的认识，就是要做强基础平台、分享落地的场景、提炼共性需求、共同编制用例。这里面既涉及对基础平台本身的建设，也涉及面向领域这样一些需求的分析和用例的开发。

成员的愿景大概可以凝聚到以下几个地方。第一个聚焦于区块链资产端的应用，同时兼顾在资金端探索；第二个是构建满足共性需求分布式总账技术平台；第三个要精选落地的场景，针对这些场景来开发应用解决方案；第四个就是一些成果中作为基础代码的部分是开源的，而作为解决方案的部分会在成员之间进行共享。

已经开展的工作包括这样几个部分，一个是自身的建设，包括组织成员单位进行交流，也包括发展这些观察员的单位。此外分四个专题开展专题的研究，包括技术，主要是对现有的大平台进行选型，对现有的大联盟的运作模式和定位进行一些初步的分析，这是技术评估部分。底下是业务，业务主要是联盟各成员之间共享或分享各自的业务场景，然后在分享的基础上争取提炼出共同需求的共性，尤其是对于底层平台需求的共性。

在法律和合规方面，ChinaLedger 也包含一个法律方面专题的研究，这些研究不仅仅是联盟成员在进行，观察员也有参与。此外，联盟也持续地对海外相关的发展进行跟踪。

在 2016 年 9 月，联盟发布了《 ChinaLedger 技术白皮书》（以下简称《白皮书》）。技术白皮书的发布，标志着 ChinaLedger 对下一步工作的推进和目标平台的研发已经形成了系统性的观点并且在联盟成员范围内就这些观点达成了明确的共识，这是一个非常重要的节点性事件。

A.3.3 《 ChinaLedger 技术白皮书》解读

《白皮书》建立在对我国资本市场需求的深刻理解之上，《白皮书》在显著的位置采用大量篇幅，阐述了 ChinaLedger 对我国资本市场应用区块链技术的需求。

在对业务需求进行深入分析的基础上，《白皮书》提出了在我国资本市场应用分布式总账技术的推进顺序的“四阶段”建议，即“场外业务→交易后业务→业务沙箱→国际化业务”。这样的推进顺序建议，是经过反复斟酌和慎重考虑的。

《白皮书》建立在对区块链平台最新技术的调研评估基础之上。《白皮书》在形成的过程中，结合我国资本市场的具体应用需求，对主流区块链技术平台进行了深入的考察。进行考察的对象包括比特币、以太坊、比特股、Ripple、HyperLedger 和 Corda。这六大技术体系中，前四个是技术平台 + 数字货币 + 社区三位一体的，后两个是技术平台 + 业务（特别是金融业务）场景的。这样的选择，兼顾了实际应用背景的丰富程度和技术本身的成熟程度，使 ChinaLedger 的设计理念和技术决策建立在更加稳妥和安全的基础之上。

ChinaLedger 对评估对象的考察，分为如下十个基本维度，分别是：

1）领域适用性；

2）场景适用性；

3）计算能力完备性；

4）架构分层合理性；

5）共识达成机制与效率；

6）计算与存储效率；

7）隐私及特权机制；

8）原生数字货币的意义与必要性；

9）开发与技术支持；

10）未来发展潜力与动向。

这些基本维度，是任何一个团队在基础协议层面基于开放技术资源打造分布式账本新技术平台时都必须要考虑的因素。被考察技术体系在这些维度上的表现和得失，对于 ChinaLedger 有着重要的借鉴意义。

《白皮书》确立了“借鉴→改造→自主开发”的三阶段发展战略。

ChinaLedger 最核心的部分是其第 4 节“技术路线”（见附录 B.4 节）。这一部分是对 ChinaLedger 重大技术决策和战略安排的一个总的叙述，其中提出的“借鉴→改造→自主开发”的三阶段发展战略，体现了 ChinaLedger“立足本土、实事求是、有所作为”的基调。

“借鉴”，是指在基础账本、合约语言和虚拟机层面首先全面借鉴成熟的技术平台，在其上直接开发 POC 应用。所谓成熟，应包含以下几个主要特征：

1）支持多种资产记账；

2）支持复杂业务逻辑的表示；

3）支持引入中心化要素实现隐私和特权等机制；

4）经历过复杂利益格局和安全格局下的博弈考验；

5）支持联盟链部署、性能有进一步优化的空间。

在借鉴阶段，ChinaLedger 将专注于在智能合约层面开发面向 POC 场景的应用逻辑，对被借鉴的技术平台总体上采取版本跟随策略，不做永久性分叉。

“改造”，是指在特权机制、隐私机制、资源控制机制等方面打造标准化的智能合约模板，支持国内资本市场共性的本土化需求，特别是合法依规履职运营方面的需求。《白皮书》对利用智能合约模板来达成合法依规目标的做法，在技术上是可行的，在战略上是主动的。所谓技术上可行，就是这几项需求在技术上恰好都可以通过标准化的智能合约模板来实现，无须在基础账本、合约语言和虚拟机层面大动手术，因此可以在基础平台版本不动或微调的情况下，保证智能合约模板开发的持续有效性。所谓战略上主动，就是一旦 ChinaLedger 联盟的发展战略进入“自主开发”阶段时，只要继续有效支持原有的合约语言，即可在新的基础平台上全面支持已经开发好的智能合约模板，保证基础平台的无缝迁移。

“自主开发”，是指在对区块链技术有了更加深入的了解，对国内资本市场的法律监管环境和业务需求有了更加透彻的把握的前提下，按照国家有关的信息安全标准、区块链基础协议标准及资本市场近期与未来业务所需的基础平台特性，由本土团队自主完成并 100%

掌控的含基础账本、虚拟机、网络互联、密码学函数库等在内的完整、优化的技术平台。ChinaLedger 联盟期待这个自主开发的版本高度体现中国特色，广泛凝聚业界共识，为国内区块链应用领域特别是资本市场所普遍采纳，并产生与我国国力相称的国际影响。

《白皮书》提出了一批原创的针对性解决方案。

《白皮书》的一个重要特色就是其面对国内资本市场的法律和监管环境，提出了包含特权机制、隐私机制在内的一批原创的针对性解决方案。这部分工作，具有重要的现实意义，甚至对于其他国家的资本市场开展区块链应用，也有一定的借鉴价值。

在特权机制方面，《白皮书》的一个主要贡献，就是引入特权账户，经特权账户签发的消息，可停止特定账户、特定产品（合约）、特定市场（合约组）的交易，可通过反冲指令，纠正被错误交易指令“污染”了的价值再分配方案。

在隐私机制方面，《白皮书》提出了基于“中央对手方（CCP）”的双链隐私解决方案。这个方案的特点是：交易无关方在基于密文的 A 链上对交易行为进行背书，交易相关方通过 CCP 在基于明文的 B 链上对交易内容（含实际对手方）进行背书。行为和内容的分离，使得交易相关方的隐私依法得到保护。而中央对手方本来就被赋予与这个背书行为相当的法律职责，所以所有交易对中央对手方而言都是透明的这一点，在法律上并没有任何不妥。监管当局可通过 B 链对所有交易实行看穿式管理，因此这一项监管职能也可以在技术上完整落地。隐私的进一步分组分割，比如一条 A 链多条 B 链等架构，还可支持多机构在云化的区块链环境里建立彼此之间的“隔离墙”，在共享基础服务的同时，彼此隔离关键业务数据。

附录 B Appendix B

《ChinaLedger 面向中国资本市场应用的分布式总账白皮书》全文

B.1 综述

近年来，金融领域的技术创新不断涌现，以金融科技（FinTech）为代表的一系列影响深远的技术创新，正在改变着金融服务业的业态。其中，分布式总账技术（Distributed Ledger Technology，DTL）得到了金融界和 IT 界的普遍关注。在我国，分布式总账技术的实践者们紧跟世界潮流，依托民间数字货币的各类创新性应用方兴未艾；“主战场”上的各类传统金融机构关于分布式总账技术的各类高质量的研究、有意义的探索和尝试越来越密集，力度也越来越大；金融监管机构和 IT 产业政策管理机构也对这项技术给予了高度关注。

与分布式总账概念密切关联的另一个概念是区块链（Block Chain）。一般认为，区块链是一种典型的分布式总账，在其上可以以多边共治的方式，靠密码学原理的保证来不可更改地记录价值的产生和转移行为，以可编程的方式实现与价值有关的业务逻辑，当然更一般意义下的区块链还可以可靠地记录价值以外的其他信用状态并实现相应的业务逻辑。我们也注意到了另外一种观点，即分布式总账也可以不必是区块链，只要具备多边共治的技术手段（共识机制和防伪机制）和以价值为背景的数据内容，就都可以纳入分布式总账的范畴。在本白皮书中，尽管我们提出的框架性方案基本上还是落在区块链范畴之内，但我们将主要使用“分布式总账”这一说法。

“中国分布式总账基础协议联盟”（简称 ChinaLedger）的目标是聚焦资产端的分布式总账应用，兼顾货币端和非金融端应用，从精选的应用场景中提取出若干具有普遍性的金融服务模式，分别通过基础账本的协议 / 架构层面和应用层面的技术实现对相应业务提供完整的支撑。与“互操作型的”联盟组织不同，ChinaLedger 成员机构间基本上不存在横向的互操作

关系，更多地是统一维护一套共享的共性基础平台、各自基于平台建设自身应用系统的“资源共享型的”联盟组织。

ChinaLedger 成立以来，组织成员单位系统交流了我国资本市场的法律与监管环境、典型业务场景的功能需求和分布式总账技术应用的推进顺序，认真评估了建设 ChinaLedger 可采纳的技术架构、可选择的技术资源和需要解决的关键技术，广泛参考了海外推进分布式总账技术在资本市场应用的最佳实践。在此基础之上，我们制定了此白皮书，作为推进 ChinaLedger 下一步工作的纲领和依据。

B.2 业务需求与推进顺序

本节聚焦于国内资本市场，考察在“资产端”引入分布式总账技术的必要性、可行性和先后顺序问题。

B.2.1 “资产端”的具体范围

分布式总账技术最初发源于数字货币领域，目前已可提供货币服务、价值服务、信用服务，形成外延递进扩大的三个“圈”。仅货币服务所构成的领域，我们称之为“资金端”；除货币服务之外的其他价值服务所构成的领域，我们称之为“资产端”；除价值服务之外的其他信用服务所构成的领域，我们称之为“非金融端”。事实上，资金端、资产端和非金融端所对应的监管环境和市场业态，根本上也是不同的。

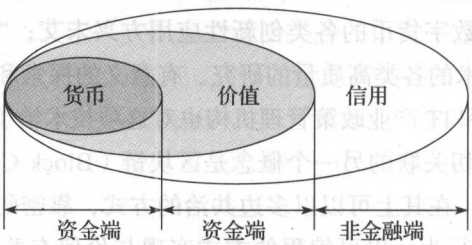


图 B-1 资产端的具体范围

具体到“资产端”，其外延包括一切具有价值属性的非货币的财产，大致可分成如下四类。

1）有价证券类：股票、债券、基金、期货、期权、资产证券化产品、结构化理财产品、非标私募产品等。

2）大宗资产类：土地（地契）、房屋（房契）、大宗商品（仓单）、贵重奢侈品、艺术品等。

3）权益类：矿业权、碳排放权、数字版权等。

4）积分类：各种奖励积分、折扣抵用券等。

其中，第 2 至 4 类业务基本上属于“场外业务”，第 1 类业务中，既有适合“场内”开

展的竞价类业务，也有适合“场外”开展的报价类业务。交易所、外汇交易中心的固定收益类产品和仓单类产品，中证协下属的机构间报价平台和各地的区域性股权交易市场均属于“场外业务”。

一般来说，“场内业务”在交易环节会涉及比较复杂的时序逻辑关系和匿名的交易对手方，在清算环节会涉及中央对手方净额担保交收制度，还会涉及复杂的即时行情披露，流动性好，对实时性能和系统可用性要求高，对交易差错的纠正比较困难，因此其对全局性风险的容忍度极低。特别是，“场内业务”各类职能角色已通过法律、部门规章和业务规则等形式确定了相应的特许经营主体（例如：交易场所、登记结算机构、市场经营机构等），除非修法，否则此类职能角色几无可能发生变化。

相对而言，“场外业务”在交易环节中大体上具有相对简单的时序逻辑关系，交易对手方是相对实名的，在清算环节往往会采用非担保交收制度，流动性较差，交易相对离散，因而对实时性能的要求不高，对交易差错的纠正也相对容易，因此相对于场内业务而言其对全局性风险的容忍度较高。场外业务各类职能角色的定义层级相对较低，对其进行变动涉及的流程也相对简单。

在我国从事场外业务的机构大都使用自建的技术平台。相对于平台的实际业务负载而言，平台的开发、维护、运行成本因必要的冗余配置而在全局来看略显重复，跨机构的平台整合虽然从经济上是合理的，但所涉及的机构之间面临着信任基础不足的问题。新设立的从事场外业务的机构在平台的技术路线选型上也面临着是否选择“云服务”和“分布式总账”的问题。

B.2.2 业务功能

基于分布式账本开展“资产端”业务，大约包含如下几类功能。

1. 发行

这是价值通过非挖矿的形式从无到有、从聚到散的过程，可能包含两种形态：一种是确权发生在链下，当确权完成时应将相应资产份额转移到链上进行登记托管。这种业务类型相对简单，除资产份额数据之外，登记托管职能机构对确权相关的电子文档资料的数字签名也应放到链上备查。另一种是确权本身通过链上“认购”智能合约的业务逻辑执行过程产生，其本身就是分布式应用的一部分。这种业务类型较为复杂，在国内资本市场的场内业务中会经常见到。如果不仅认购份额、连“发行价”也要以“众人参与、众人见证”的可编程的方式来确定，那么相应的智能合约将更为复杂，目前国内资本市场无论场内还是场外都还没有先例。

发行后的资产总份额一般情况下是守恒的，但在分红派息、配股等场合，资产也会发生并非由交易 / 转让导致的变动（公司行为）。这些公司行为也属于“发行”的一部分。

ChinaLedger 应能支持通过智能合约在链上“认购”的发行方式，支持发行后通过公司

行为引起的资产变动。

2. 交易 / 转让

交易涉及资金和资产的双向流动。目前的做法有如下两点。

1）资金在链下、资产在链上，资金通过分布式总账平台与资金系统之间的“支付网关”来进行。在真正的支付发生之前，资金通过链上开设的“虚拟头寸”来进行记账管理。

2）资金、资产都在链上，交易即支付。

无论哪种情况，都必然会涉及分布式总账对交易完整性的管理，也就是说，一笔交易要么资金、资产之间的双向流动均已完成，要么均未完成，不应出现一方已完成、另一方未完成的情形。

普通的资产交易不允许“买空”和“卖空”。在分布式总账中，这将涉及有关透支控制的余额检验。

除非明确规定交易信息公开，否则交易信息是交易双方隐私的一部分，分布式总账中应避免除交易双方之外的普通账户看到或轻易就能推测出这类隐私数据。

场内交易的主要形式是连续竞价。连续竞价的时序逻辑是“订单驱动、价格优先、时间优先”。场外交易的主要形式是“报价驱动、双边报价、点击成交”。部分产品具有远期的反向操作，比如股权 / 债券的“质押式回购”。

ChinaLedger 应能同时支持资金和资产在链上的双向流动，支持交易完整性，支持买空卖空控制，支持交易数据的隐私保护。ChinaLedger 应优先支持报价驱动的场外交易模式，之后再酌情考虑支持订单驱动的场内交易模式及“回购”模式。

3. 结算

直接使用基础账本记账的场合，交易即结算。

使用智能合约处理交易的场合，可在智能合约内部完成净额轧差，在回到基础账本的环节实施交收。

在使用分布式总账平台仅处理场内交易业务的交易后环节的场合，交易信息需要依赖一个前置于分布式总账平台的消息基础设施（待建设）来汇集，由分布式总账平台在基础账本或智能合约的层面来完成相应的清算和结算动作。

如果担任“中央对手方”的职能机构由法律明确规定，那么分布式总账平台不应改变这一职能。

ChinaLedger 将优先考虑实时使用智能合约进行净额轧差、在回到基础账本的环节批量实施交收的业务模式。

4. 交割 / 行权

数字权益的行权可直接在基础账本上执行过户。

实物权益的交割一般在线下执行，但使用分布式账本技术可以依据业务逻辑直接变动权益凭证（如电子仓单）的权限状态，提高线下执行的安全性和自动化水平。

ChinaLedger 将支持使用分布式账本平台进行数字权益的行权，为实物权益的交割提供更好的保障。

B.2.3 监管功能

我国资本市场的法律、法规、行政规章和业务规则赋予各类带有监管功能的主体各自依法合规行使监管职能及特定的操作特权，具体如下。

司法部门有依法冻结指定账户交易的特权。

监管部门有依法“看穿”所有账户的开户、交易、持有数据的特权。

交易所有依法对特定交易产品实施“停牌”、对特定市场实施“停市”、对显失公平的交易予以取消、对达到特定风控警戒标准的仓位予以强行平仓、强行减仓的特权。

登记结算机构有依法对特定的已达成交易实行暂缓交收或拒绝交收的特权。

目前尚未看到有分布式账本平台完整支持这一系列含有明确“中心化”特征的特权。ChinaLedger 将支持这些特权在分布式账本平台上“落地”。

B.2.4 推进顺序

根据对国内资本市场各类不同交易结算机制的法律定位和业务特点的分析，结合它们在分布式总账平台上落地的技术难度，ChinaLedger 建议国内资本市场的分布式总账应用按如下顺序来推进。

1）**场外业务**：场外业务是与分布式总账业务场景契合度最高的业务，也是法律监管环境最容易随着技术进步调整适应的业务。对于已经存在的场外市场，可以尝试在单市场利用分布式账本技术去单独支持个别新推出的业务，也可以联合多市场利用分布式账本技术进行云化整合，在云化整合的场景下要注意在技术上保障市场间交易信息的有效隔离。对于正在筹建的场外市场，则建议直接把业务建立在分布式账本技术之上。

2）**场内业务的交易后业务处理**：这是资本市场最核心的业务之一，目前各相关机构（证券公司、交易所、登记结算公司）花在交易后清算结算和对账处理上的时间很长，采用分布式总账技术可以大大缩短这一时间，同时也为相关机构节约大量 IT 开支。

3）**业务沙箱**：在指定的资本市场业务“特区”内，封闭运行由分布式账本技术支持的场内业务，在鼓励技术创新的同时，审慎观察市场表现，严格控制风险范围，使之不会扩散。

4）**国际化业务**：随着世界各国对分布式总账技术认知的不断加深和应用的不断推进，我国的资本市场在今后的对外开放中若要与境外资本市场相对接，那么使用分布式总账技术促进对接将成为一个可能的选项。

上述推进顺序还与央行数字货币工作的推进进度密切相关。如果央行数字货币工作得到提速，那么可能会进行一些局部调整。

B.3 技术选型评估

为满足前述业务需求，ChinaLedger 对目前世界上最有影响的六大分布式账本技术体系（比特币、以太坊、比特股、Ripple、HyperLedger 和 Corda）进行了深入的考察和评估。需要指出的是，其中前四个技术体系是平台、货币、社区三位一体的，后两个技术平台是“纯平台的”，但我们的评估仅针对平台，不涉及货币和社区。

考察评估的维度涉及领域适用性、场景适用性、计算能力完备性、架构分层合理性、共识达成机制与效率、计算与存储的效率、隐私及特权机制、原生数字货币的意义与必要性、开发与技术支持、未来发展潜力与动向共计 10 个方面。

B.3.1 领域适用性

比特币账本底层数据结构拥有的唯一一个价值字段用于描述比特币价值创造和转移的面额。换句话说，比特币技术体系如要移作他用，那么其只能提供单一标的资产的登记和转移。如要同时支持多种标的资产共处和交易，则还需进行相应的改造。

以太坊、比特股、Ripple、HyperLedger 技术体系都能同时提供多种标的资产（含数字货币）的登记和转移服务，天然支持数字货币和数字资产在同一个区块链上共处，这对于构建具有资产交易业务逻辑的资产端应用来说是更加方便的。此外，除原生数字货币之外，由外部注入的数字货币（比如代币等）在技术处理上与普通的数字资产无异。外部注入的货币与原生数字货币之间的汇兑，其技术实现方式也与资产交易类同。

在非金融端，各技术体系一般都在底层数据结构中提供一个文本类型的信息字段，可供信息提供方签名分发，作为“经签发方确认的消息”，间接提供非金融领域的信用服务。在以太坊和 HyperLedger 技术体系中，经签发方确认的消息还可触发智能合约执行相应的动作。

B.3.2 场景适用性

根据分布式总账的技术特点，一个应用场景的参与方，既是业务的参与主体，同时又是其分布式总账本身的运营和见证主体。一般可根据参与方加入应用场景是否需要获得许可，把场景分为“非许可的”和“许可的”两大类。在区块链社区中也把“非许可的”场景称为“公有链”，把“许可的”场景细分为“私有链”和“联盟链”。私有链是单边治理的业务生态，联盟链是多边共同治理的业务生态，公有链是整个社区共同治理的业务生态。

比特币、以太坊、比特股、Ripple 都是通过自身社区共治共享的公有链体现其技术体系对公有链场景的适用性，鉴于公有链社区人员组成的复杂性和博弈的高度对抗性，能够在公有链环境下生存下来的分布式总账技术平台，在安全上是经得起考验的，不加改造或略加改造作为联盟链或私有链部署也具有可行性。

HyperLedger 目前的设计是以联盟链为出发点，但是其白皮书强调每个模块（包括身份认证和共识算法及数据库协议等模块）的可插拔性，兼顾了今后作为公有链的可能性。Corda

目前已公布的资料较少，但也可以从中清晰地看到其非公有链的取向。

B.3.3 计算能力完备性

价值可编程是分布式总账技术的一个重要的本质属性，直接决定平台对业务逻辑的表达能力，具体体现在“智能合约”上面。比特币内置脚本的表达能力是极为有限的。Ripple 目前不支持智能合约。Bitshares 的智能合约在运用上有很多限制，并且不能自定义。以太坊和 HyperLedger 支持智能合约且可达到“图灵完备”程度。

B.3.4 架构分层合理性

目前，业界对于分布式总账的基础协议栈结构并无统一共识，各技术体系做法不一。无论是从快速构建应用角度来说，还是从与分布式账本之外的技术资源进行整合的角度来说，甚至从未来占领标准化制高点的角度来看，架构分层的合理性都是一个应该引起高度关注的议题，架构分层朝着更合理方向的每一次改进，既体现了业界对分布式账本技术架构理解的深化和运营理念的升华，往往也酝酿着新的商业机会。

ChinaLedger 期待的分布式账本技术体系的架构，应该体现三类不同性质的节点（记账端、验证端、客户端），五个不同的协议栈层次（网络通信、基础账本、共识、智能合约、应用）及四个不同的管理要素（身份、策略、数据、过程）。从长远来讲，有些共性技术（如 P2P 通信和执行智能合约的虚拟机环境）或许应该交给更合适的主体来做。从现实情况来看，六大体系中，HyperLedger 的架构分层显示出了更多的包容性和更大的弹性，其各组成模块有灵活的可插拔性，便于支持各种法律和监管环境下分布式账本技术的落地，各模块间的相互关系也较为合理。该架构的优点值得 ChinaLedger 学习和借鉴。

HyperLedger 架构的总体布局如图 B-2 所示。

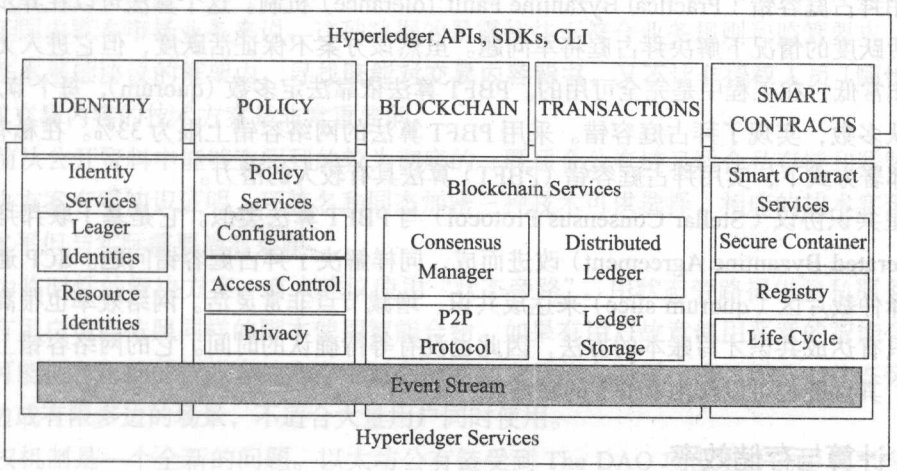


图 B-2 HyperLedger 的架构总体布局

B.3.5 共识达成机制与效率

目前可供选择的共识达成机制有工作量证明机制、股份证明机制、股份授权证明机制、Ripple 共识机制和实用拜占庭容错机制等。

工作量证明机制（Proof of Work，PoW）的优点是可以达到完全去中心化，节点自由进出。缺点是消耗大量的资源，达成共识的周期较长，不适合在资本市场的“主战场”上应用。而且从统计的角度上讲是需要 6 个或以上的确认才能认为是明确确认且不可逆的。其网络容错的上限是 50%。典型应用是比特币和以太坊。

股份证明机制（Proof of Stake，PoS）已有很多不同的变种，但基本概念是产生区块的难度应该与对应节点在网络里所占的权益（所有权占比）成反比。PoS 的优点是在很大程度上缩短了共识达成的时间。它的网络容错上限也是 50%。典型应用是点点币（Peercoin）和未来币（NXT）。以太坊计划在未来使用的 PoS 算法叫 Casper，验证人数最多 250 人，并且区块一旦达到最终状态（final）就完全不可伪造。

股份授权证明机制（DPoS）。它其实是 PoS 的变种，由全部节点记账变为选出代表节点记账。当使用去中心化自治公司（Decentralized Autonomous Company，DAC）这一说法时，每个股东按其持股比例拥有影响力。每个股东都可以将其投票权授予一名代表。获票数最多的前 N 位代表成为验证者，并按既定时间表轮流产生区块。它的网络容错上限同样也为 50%。典型应用是比特股（Bitshares）。比特股需要等待半数以上的验证者确认才认为区块不可逆。

瑞波共识机制（Ripple Consensus）。瑞波共识机制设定了一组特殊的节点列表，只有在这个列表中的节点才是有效的验证者。这种机制达成共识的效率非常高，并且只有达成共识的区块才会写入账本。因此写入即有效，无须等待确认。为了达到高可靠性，只有当 80% 的验证者都同意交易才算有效，即网络容错上限为 20%。

实用拜占庭容错（Practical Byzantine Fault Tolerance）机制。这个算法可以在异步网络中不保证活跃度的情况下解决拜占庭将军问题。虽然该方案不保证活跃度，但它进入无限循环的概率非常低，在工程中是完全可用的。PBFT 算法依靠法定多数（quorum），每个节点一票，少数服从多数，实现了拜占庭容错。采用 PBFT 算法的网络容错上限为 33%。在私有链 / 联盟链的部署方式下，实用拜占庭容错（PBFT）算法具有较大的潜力。

恒星共识协议（Stellar Consensus Protocol）与 PBFT 算法类似。它是基于联邦拜占庭协议（Federated Byzantine Agreement）改进而成，同样解决了拜占庭容错问题。SCP 通过节点自行选择仲裁片区（quorum slice）来达成共识，增减节点非常灵活。网络效率也很高，采用的也是只有达成共识才写账本的方法，因此也没有等待确认的时间。它的网络容错上限同样为 33%。其性能也可以作为 PBFT 的参考。

B.3.6 计算与存储效率

目前运行着平台 + 货币 + 社区三位一体的公有链上，我们所观察到的计算与存储效率受

到了很多因素的制约，而且计算与存储之间存在目标冲突，并非技术上完全不能实现更高的效率，所以参照的意义并不是很大。

要实现分布式账本技术的大规模应用，在计算与存储方面做进一步的性能优化是不可避免的。目前可选择的技术方案共有三种方式，具体如下。

一种方式是以定期保存的**账本快照**（snapshot）当作整个网络共同认可的状态。按照这种方式，全量历史记录有可能回退到云化甚至中心化存储，这在公有链上相当于是在安全性和去中心化上做出了一定的妥协。我们也可考虑以诸如 IPFS 等分布式文件存储方案来降低中心化存储的风险。

另一种方式是**分片处理**（sharding）。这种方式主要是出于解决计算性能问题的考虑，但是也兼顾了缓解存储问题的需要。总体思路是，每个节点只处理一部分（比如一部分账户发起的）交易，从而减轻节点的计算和存储负担。但是这种方式也会带来新的问题，如在复杂时序逻辑下数据的一致性、交易的原子性和交易的相互依赖对性能的影响等。

第三种方式名为**状态旁路**（State Channels）。这种策略是保持底层的区块链协议不变，通过改变协议使用的方式来解决扩展性问题。在这种策略下，分布式账本上可见的只是粗粒度的“批发”，可以类比出入备付金操作，而真正细粒度的双边或有限多边交易明细，则不作为“交易”记录在分布式账本上，而仅仅作为有争议事件发生时备查的“信息”单据，通过状态旁路的方式“曲线”执行。比特币体系下的“闪电网络”是在比特币脚本逻辑表达能力受到限制的情况下不得不借助“精巧”的设计实现的事实上的状态旁路。在以太坊体系下，借助智能合约的丰富表达能力，状态旁路的实现即可得到大大简化。

上述三种优化思路并不是彼此排斥的，而是可以组合使用的。

B.3.7 隐私及特权机制

目前分布式账本上的交易数据（包括交易内容、发送方和接受方的地址）都是公开可见的，对于国内资本市场业务来说，这种数据的暴露往往不符合业务规则和监管要求。若要在分布式账本基础协议的框架内，寻找**既能对交易内容背书、又不让非授权人员（哪怕是背书者）获取交易内容**的技术方案是非常重要的。

目前从公开资料中能够查阅到的较为彻底的、既适合公有链又适合私有链和联盟链的密码学解决方案有零知识证明、环签名和同态加密三种技术可供选择，相应的技术实现虽已接近可用水平但与实际需要尚有差距。

作为临时性的解决方案，有人建议使用“状态旁路”。用状态旁路提供隐私服务，其前提是所有用户均应按照同样的脚本使用智能合约。如果有用户故意使用恶意的智能合约，那么其有可能把在正确的智能合约内存中解密的明文交易信息泄露出去。因此，状态旁路只适合于双边或有限多边的场景，不适合大量用户同时使用。

特权机制是一个全新的问题。以太坊公有链受到 The DAO 攻击事件的影响至今仍在持续，类似事件如果出现在资本市场的“主战场”上那么后果将是不堪设想的。但在评估过程

中，我们没有发现所考察的技术体系在这一问题上有可供选择的解决思路。

B.3.8 原生数字货币的意义与必要性

目前几乎所有部署在公有链上的分布式账本技术体系里都有原生的加密货币。这些加密货币所具有的共同特点都是没有中心化的发行方，可以在其对应部署的公有链上自由流转。这些原生数字货币的用途包括：支付手段、汇兑手段、抵押手段、激励手段、权益证明和资源控制等。

随着一套分布式总账技术体系从公有链平移到私有链 / 联盟链场景，前面所说的关于原生数字货币的很多用途都会被消解掉，锚定法币的代币会取代一部分功能，因此在私有链 / 联盟链的建设过程中，“去币化”已成为一道标配的工序。但是，“权益证明”和“资源控制”这两个职能，即使到了私有链 / 联盟链场景，也仍然有存在的必要性。原生数字货币或许仍不失为在私有链 / 联盟链场景下履行这两个职能的一种单纯的计量和调节工具。

B.3.9 开发与技术支持

据不完全统计，比特币的核心代码库、以太坊的 Go 语言核心代码库、Ripple 的核心代码库、比特股 2.0 的核心代码库等均已进行了多次升级，超级账本的项目 Fabric 与 Sawtooth Lake 都进行过升级。应该说这些平台的核心代码都积累了大量的在线升级经验。

相比之下，智能合约是契约，更是程序，是程序就难免会有升级的问题。那么如何处理智能合约的升级，如何保证智能合约的状态和逻辑在升级前后有序衔接，如何保证智能合约的升级和平台的升级相得益彰而不是互相掣肘，这些也是全世界都在面对的艰难挑战。

比特币、Ripple、比特股的核心代码主要由 C++ 编写。HyperLedger 的 Fabric 项目核心代码主要由 Go 编写。HyperLedger 的 Sawtooth Lake 项目核心代码主要由 Python 编写。

以太坊的黄皮书是对以太坊的形式规范描述（formal specification），即用计算机科学的形式语言来描述的以太坊系统的规范。参照黄皮书的规范可以用各种编程语言实现客户端。目前以太坊已经包含了经过大量安全审计的 Go 语言、C++ 语言和 Python 语言实现的 3 个客户端，另外 Java 和 Ruby 的客户端也在开发中。

除 Corda 之外，其他平台都是开源项目，项目文档比较丰富。但最近 Corda 发布了非技术白皮书，使业界对其理念和技术路线有了较多的了解。

B.3.10 未来发展潜力与动向

比特币技术体系正在新的升级计划的引领下寻求新的突破。尽管其典型公有链、单一标的资产及 PoW 共识机制等特质决定了在金融领域获得广泛应用存在很大难度，但我们还是希望看到新的升级计划给比特币技术体系带来新的跨越。

Ethereum（以太坊）制订了清晰的进一步开发的路线图，具体涉及浏览器、共识机制、虚拟机和可扩展性方面的重大改进。此外，以太坊还在积极探索满足金融行业需求的各种技

术路径，这些探索涉及隐私保护、吞吐量及功能更强且更加可靠的智能合约等方面。

Ripple（瑞波）现阶段主要致力于与银行合作，解决汇兑类问题，而对更全面的应用于金融领域所必须考虑的隐私保护、可扩展性、合规性等问题尚未有明确的解决思路和研究计划。

Bitshares（比特股）尚没有形成长期的、稳定的核心开发团队，其创始人 BM（Daniel Larimer）也于 2016 年 4 月表示，他将逐步淡出比特股的开发。因此，比特股未来的发展前景并不明朗。

Corda 和 HyperLedger 分别由两家联盟组织发起建设，联盟的成员机构主要来自金融领域和 IT 领域。它们将聚焦于分布式总账技术在金融行业的运用，重点放在私有链、联盟链上。从已披露的信息来看，两个联盟均提出了一些有创意的设想，后续能否及如何实现这些设想是 Corda 和 HyperLedger 发展的关键。HyperLedger 是一个支持智能合约的、底层可插拔的通用协议框架，其项目多有很强的金融背景，包容性相对较强，又有很多金融领域传统服务商参与加盟，组件模块的不断丰富是毋庸置疑的。Corda 对数据的隐私性、合规性和监管需求的考虑明显更接近金融业务主战场的视角，不排除会走一条与众不同、直达金融业务本质的道路。

B.4 技术路线

B.4.1 领域、场景和运营模式选择

ChinaLedger 将为在国内资本市场推进分布式总账技术应用提供符合我国国情、适应我国法律与监管需要的基础平台，聚焦资产端应用，兼顾资金端和非金融端应用。

鉴于国内资本市场分布式总账技术应用场景更多地呈现出“同质化、小生态”的特点，ChinaLedger 将针对这一特点积极探讨带有隔离墙机制的“云化”解决方案。

针对国内资本市场的现实环境，ChinaLedger 优先考虑提供联盟链解决方案，在联盟链上忠实体现和反映国内资本市场职能机构的业务范围和职能边界。在运营模式的设计上，支持技术层面的专业化运营服务和业务层面的自主化运营服务，既互相剥离，又有机结合。

B.4.2 平台策略

根据前面的技术选型评估，我们大致可以得到关于平台的一组基本的策略。

1. 借鉴

分布式账本技术的核心就是对算法的充分信任，而在算法代码开源的条件下，通过在广泛应用和反复博弈中取得公众信任，是对一个分布式账本技术体系强安全性的最好诠释。因此从近期来看，ChinaLedger 的分布式账本基础平台不可能凭空产生、从头做起，而必须首先选择合适的开源分布式账本技术体系作为主要借鉴，在其基础上搭建符合我国国情、适合

我国法律与监管需要的基础平台。

从前面的业务需求分析中不难看出，高安全性、强表达力、多资产类别、良好的未来发展潜力和性能优化前景，去除原生数字货币的副作用小，是国内资本市场对基础账本选择的基本要求。而在我们所考察的六大技术体系里，最有借鉴价值的，是以太坊和超级账本两个平台。

但随着技术的发展，新的平台也在不断出现。另外从长远考虑，自主可控底本的研发也应提上日程。因此，初期的借鉴不是最终目的，ChinaLedger 还必须对长远的底本选择做出战略性安排。

2. 改造

从前面的业务需求分析中不难看出，ChinaLedger 为适应国内资本市场要求而拟在后续对分布式账本基础平台进行的主要修改工作，核心在于中心化的“特权”机制的引入、“隐私”机制的实现、原生数字货币的去除和业务处理性能的优化。它们几乎都不约而同地集中在智能合约层面。我们可以判断，合约语言的未来趋势一定是跨平台的，即使是在同一平台下，合约模板也完全有可能对基础账本层的细微版本变化无感。如果再加上技术实现方案上的刻意努力，这就意味着可以兼顾近期的借鉴和长远的自主开发，我们在合约层面拟开展的改造工作不仅在近期会快速得到应用，在未来也会得到最大限度的复用。因此，ChinaLedger 的目标是：在智能合约层提出一套“合约模板”，尽量把所有的改动都封装在“合约模板”中，这样就可以在某种抽象意义下同时既支持近期以借鉴为主的基础账本，又支持远期以自主开发为主的基础账本。

3. 自主研发

分布式总账技术具有很强的安全性，资本市场又是国民经济的命脉。从长远战略的考虑来看，ChinaLedger 应在充分掌握底层账本核心技术的前提下，自主开发出一套含基础账本在内的整个分布式总账技术体系。

涉安全技术模块的自主掌控、涉资本市场模块的合法合规、架构的充分模块化和架构组件的充分可插拔是 ChinaLedger 未来自主研发的分布式总账技术体系的设计所应遵循的基本架构准则。

在数据的加解密算法和摘要算法上，ChinaLedger 应支持国密算法。

以上关于平台选择策略的借鉴、改造和自主研发三原则，可用图 B-3 来表示。

B.4.3 共识机制选择

根据选型分析的结论，PBFT 所能达到的性能指标在各类共识机制中名列前茅。据我们了解，目前，在以太坊平台和超级账本平台上都有引入 PBFT 共识机制的尝试。ChinaLedger 内部也进行了在以太坊平台上引入 PBFT 共识机制的测试。应该说，引入 PBFT 已经不存在本质上的技术障碍了。

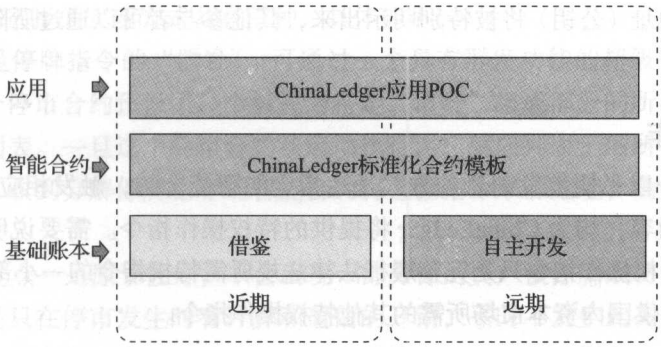


图 B-3 平台选择策略的借鉴、改造和自主研发三原则

但是也要看到，PBFT 不适用于节点可动态加入的场景。针对这种情况，ChinaLedger 还将深入研究 PBFT 的替代方案。

B.5 特权方案

我国现行法律制度赋予司法机关和特定金融机构在金融业务中行使某些职能的特权[⊖]。比如，业务规则不可以对抗司法冻结；监管机构可以根据工作需要，按程序查看某些涉隐私数据；交易所可以对从事杠杆交易的投资者账户实施强行平仓操作，可以对特定产品进行临时停牌，可以对特定市场实行临时停市等措施；登记结算机构可以对显失公平的交易结果采取暂缓交收乃至取消交易等措施；等等。这些措施具有鲜明的中心化色彩，虽不被社区认同，但却是资本市场得以正常运行的根本保证。

在赋予了私钥对操作个人资产独一无二许可作用的各分布式账本技术体系及其基础协议当中，在基础账本层面均未见到这种合法支配技术意义下属于他人名下财产的中心化技术安排。日前以太坊公有链受到针对智能合约 The DAO 的攻击，无论是暂停交易、交易回滚还是取消交易，这些在传统金融机构非常经典的应急手段，在以太坊体系内都无法实施，而只能采取硬分叉。这一方面源于社区无政府主义势力的阻碍，另一方面也源于必要的中心化特权机制的缺失。

如何为特定有权机构行使特权职能提供技术上的方便而又不引起安全上的问题，是分布式总账技术走进金融“主战场”所必须解决的问题。本节将介绍 ChinaLedger 关于一些特权机制在分布式账本上实现的技术方案。

B.5.1 特权账户

ChinaLedger 平台上将设立特权账户，包括但不限于司法账户、监管账户、交易业务操作账户、结算业务操作账户等。

⊖ 本白皮书中“特权”一词对应的英语是“special permission”而不是“privilege”。

特权账户的地址（公钥）将被特别明示出来，其他参与者可以通过所附签名验证操作指令是否为其所发。

B.5.2 特权操作

特权操作指令以私钥签发消息的方式送达指定的智能合约，触发相应的特权操作动作。以下各节所述的内容，均为 ChinaLedger 将提供的特权操作指令。需要说明的是，目前本白皮书中所介绍的特权操作指令只是完整履行法律法规所需特权指令的一小部分。ChinaLedger 还将继续研究并提供国内资本市场所需的其他特权操作指令。

1. 冻结 - 解冻

冻结指令由司法账户签发。指令中含有冻结类型信息、被冻结资产对应的智能合约地址和被冻结账户的地址。

冻结指令生效后，特定账户向智能合约发出的后续指令将被拒绝执行。除非收到解冻指令，冻结指令将持续生效。

解冻指令亦由司法账户签发。指令中含有被冻结资产对应的智能合约地址和被冻结账户的地址。

在收到解冻指令之后，被冻结的特定账户恢复正常权限状态。

在需要区分单边冻结（禁止价值转出）和双边冻结（既禁止价值转出也禁止价值转入）的场景，ChinaLedger 将增加双边冻结指令。考虑到价值转入的指令是将对手方账户信息加密传输的（见后文 B.6 节关于隐私方案的部分），对双边冻结指令的执行，需要具有对手方账户信息解密权限的中央对手方账户的配合。

2. 停牌 - 复牌

ChinaLedger 考虑了每个智能合约对应单项资产交易的情形。这时，对单项资产交易的停牌，就等价于对智能合约“刹车”。

停牌指令由交易业务操作账户签发，指令中含有停牌参数和被停牌资产对应的智能合约地址。

停牌指令生效后，相应的智能合约便不再接受除特权指令之外的任何普通交易指令，合约运行进入类似以太坊智能合约中 GAS 耗尽的状态。

复牌有自动复牌和手工复牌两种。

对于手工复牌，需要由交易业务操作账户签发复牌指令。指令中含有与被停牌资产对应的智能合约地址。在收到复牌指令之后，被停牌的单项资产对应的智能合约将恢复正常的运行状态。

对于自动复牌，由业主根据其自身业务规则来设定复牌时间，通过停牌参数编入停牌指令。被停牌资产对应的智能合约将于指定的复牌时间恢复正常运行状态。

3. 停市 - 恢复交易

在 ChinaLedger 中，一个“市场”可以看作是一组标的资产对应的智能合约群体。所谓

“停市”，目前可供选择的实现方式共有三种，具体如下。

第一种方式是停牌指令的“群发”，可通过一个具有群发功能的超级智能合约——停市合约来实现。这个停市合约预设了一个被视为特定“市场”组成部分的所有标的资产对应的智能合约地址的列表。一旦这个停市合约被启动执行，它就向列表上的所有智能合约地址发送分解停牌指令。收到分解停牌指令的智能合约将检验指令来源地址，确为停市合约所发则执行停牌。

这种方式的优点一是原子性好，不会出现一部分标的资产被停牌成功、另一部分停牌不成功的情况；二是只在停市发生时有计算和通信的开销，平时不发生此类开销；三是标的资产如何组成市场，只在停市合约一处维护。缺点是群发的分解停牌指令是由智能合约向智能合约发出，不再携带交易业务操作账户的签名，这样的指令安全性稍弱。

第二种方式是通过一个智能合约来集中维护市场状态，我们称这个合约为“状态合约”。正常情况下，组成市场的每个标的资产对应的智能合约在处理每一笔普通交易指令之前，均需向状态合约确认市场状态为“交易”才予以处理。需停市时，由交易业务操作账户签发停市指令，发送给状态合约，由状态合约将市场状态置为“停市”。组成市场的每个标的资产对应的智能合约在处理下一笔普通交易指令之前，如果检查到其处于“停市”状态，即可启动本身的停牌。

这种方式的优点一是无需分解停牌指令，因而也就无需交易业务操作账户批量签发的或直接免签名，它们可达到同样的安全性；二是市场状态只在一处进行维护。至于缺点，一是不仅在需要停市的情形下而且在正常情形都需要发生计算和通信开销；二是有可能无法保证停市操作的原子性；三是一个单项标的资产属于哪个市场，需要在它自己对应的智能合约中予以设定，维护工作过于分散，集中度不够好。

第三种方式是在客户端编写批量签发停牌指令的简易脚本。这种方法的优点是安全性好，标的资产如何组成市场也是在一处维护。缺点是本属于平台的功能需要在客户端编程实现，不够理想。

ChinaLedger将进一步研究上述三种方案应该如何改进。

恢复交易指令的实现方式也同停市指令一样包含三种对应的方案。收到恢复交易指令（或其分解复牌指令）后，各项标的资产恢复正常交易状态。

4. 强制划转

强制划转指令由结算业务操作账户签发，指令中包含转出方账户、转入方账户和具体划转额度的信息。这些信息具有较强的隐私性，因此和普通交易指令中的转入方账户、划转额度信息一样，需要纳入隐私保护的范围，具体详情见后文B.6节。

需进行强制划转的情形可能包括但不限于显失公平交易的回滚冲正等。

B.5.3 特权滥用问题

引入特权账户和特权操作这样一些“中心化的”机制之后，分布式账本会不会因为特权

滥用而受到损害，成为很多人担心的一个问题。从技术上如何防止特权滥用，对于这一点，ChinaLedger一直十分关注。在特权机制的设计上，也对此有所考虑，主要体现在以下几个方面。

1）**留痕**。特权操作均通过特权操作指令来进行。所有特权操作指令均需特权账户私钥签名并在智能合约中予以验证。这些带有数字签名的特权操作指令将在分布式总账中永久保存，成为实施特权操作的证据。

2）**分权**。不同的特权操作必须由不同的特权账户签发执行。特别是，负责强制划转资产的特权账户和负责停牌停市的特权账户应赋予不同的自然人。这样，即使发生了不当的强制划转资产的操作，划走的资产仍然从总体上留在价值守恒的智能合约当中。这时，只要实施紧急停牌，仍有可能追回不当划走的资产。同时，对于特权账户，均可考虑通过多重签名的方式增加安全性。

3）**授权关系**。特权用户的操作权限仅针对智能合约，暂不涉及基础账本。在基础账本上，私钥持有者支配相应账户资产的原则并未发生变化。在智能合约中，特权账户的操作权限是明示的，加入智能合约、参与智能合约中相应资产交易的参与者是知情的。在知情的情况下仍然加入，则意味着同意合约，包括同意合约中的特权安排。因此，从法律关系上看，特权用户是在被参与者授权的情况下履行特权的，也要对参与者承担滥用特权的后果。

B.6 隐私方案

在上文中，我们对交易数据的隐私特性进行了分析，提出了应对隐私数据予以保护的需求。

并且进一步分析了外界提出的与分布式账本配套的隐私保护的可能方案。通过分析可以得知保持全部去中心化特性和基于密码学最新成果的、较为彻底的隐私保护方案还达不到实用水平，而基于状态旁路的临时性隐私保护方案，难以防止通过恶意智能合约截听隐私数据。

本节将提出基于中央对手方的双链隐私保护方案。这个方案带有明显的中心化色彩，但是对于法律本来就有明确“中央对手方”职能安排的资本市场来说，这一方案又是从法律到技术的一个很自然的延伸。

由于报价驱动模式下和订单驱动模式下的隐私保护方案在细节上有很多的不同之处，而ChinaLedger优先考虑在报价驱动模式下占主导的场外市场应用分布式账本技术，因此以下仅讨论报价驱动模式下的隐私方案，关于订单驱动模式下的隐私方案我们将另行发布。

B.6.1 交易指令

在报价驱动的模式下，交易指令含有交易对手方和交易内容的信息。具体到通过智能合约在分布式总账平台上的实现，交易对手方信息和交易内容的信息既是需要分布式节点予以

见证的关键信息，同时又都是隐私信息。如果对交易内容进行加密，仅有交易对手方能够解密，那么见证者就无法通过交易内容的密文确定这笔交易是否涉嫌透支（买空或卖空），而且交易对手方必须通过链下的另外途径得知这笔交易是与自己有关的。

一个自然的解决方案就是：引入中央对手方（CCP）。交易指令用中央对手方的公钥进行加密。这就意味着，真正的余额，只有中央对手方才知道；是否涉嫌透支（买空或卖空），只有中央对手方才能判定。对交易的见证主体，和对透支与否的判断主体，在此进行了分离。引入中央对手方的交易如图 B-4 所示。

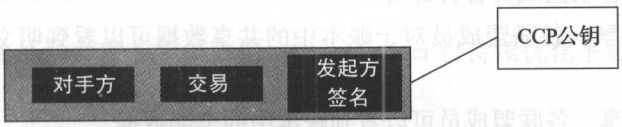


图 B-4 引入中央对手方的交易

B.6.2 双链模型

真正的余额明文，是存放在另一个分布式账本中的。为区别起见，我们把所有参与者都能访问的分布式账本记为“链 A”，把只有中央对手方和监管者能够访问的分布式账本记为“链 B”。

在链 A 上，交易指令以密文的形式出现。交易发起方将对手方信息和交易内容信息用自己的私钥进行签名，然后打包用中央对手方的公钥进行加密，发给智能合约。所有参与人均可通过分布式账本技术平台，在链 A 上见证这一笔内容被加密了的交易。如果交易是资金和资产在链上对流，那么双方都需要提交这种格式的交易指令。

中央对手方用自己的私钥进行解密后，将明文的交易指令发到链 B 上，检验交易指令中发起者签名的有效性，并获得链 B 做出的是否透支的判断。

如果透支检查正常通过，那么中央对手方将在链 B 上为双方完成交易后余额的维护。

上述手续完成之后，中央对手方将在链 B 上生成、在链 A 上向发起方发送交易确认消息。

双链模型下的交易过程如图 B-5 所示。

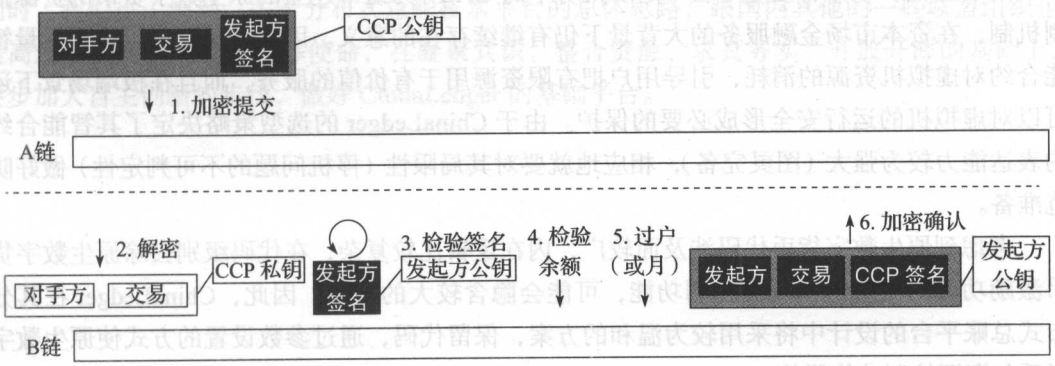


图 B-5 双链模型下的交易过程

B.6.3 看穿式监管

在双链模型下，普通参与者在链 A 上看不到交易的细节。如有需要，被法律赋予监管职能的监管者可通过开设在 B 链上的监管者账户，看到从链 A 发起的所有交易细节的明文。

B.6.4 隔离墙

一般来说，联盟体制下的分布式总账共享可以分为如下三个层面。

1）代码共享，各联盟成员各自部署。

2）账本部分共享，各联盟成员对于账本中的共享数据可以看到明文，但是对于非共享数据只能看到密文。

3）账本完全共享，各联盟成员可以看到账本中的全部数据。

在一个分布式总账平台上为多个机构提供云化服务的场景下，被服务的机构不希望其他机构看到己方的非共享数据，也不希望对方的技术性能瓶颈或其他异常影响自己的市场服务质量。这种机制在技术上称为“沙箱”。但为避免与 B.2 节中谈到的业务“沙箱”相混淆，我们将云化服务下的“背靠背”数据隔离机制称为“隔离墙”。

在云化服务的初期，隔离墙可参照本节所介绍的双链模型，通过设定一个 A 链、多个 B 链，各个 B 链之间完全不能互相访问的策略来实现。如果一个 A 链不能满足隔离要求，那么可进一步对 A 链实施分层隔离，比如在基础账本层面共享，在智能合约层面进行隔离，等等。

B.7 原生数字货币的处理

在资本市场金融服务的大背景下，初期借鉴而来的分布式总账技术体系中所建立的原生数字货币，其激励机制已经不具有继续存在的意义，因此，挖矿只有总账维护这一单一目的，且应与原激励机制脱钩。

另一方面，以某种度量单位（如以太坊中的 GAS）为纽带建立的原生数字货币的资源控制机制，在资本市场金融服务的大背景下仍有继续存在的意义。因为它不仅可以精准计量智能合约对虚拟机资源的消耗，引导用户把有限资源用于有价值的服务，而且在极端场景下还可以对虚拟机的运行安全形成必要的保护。由于 ChinaLedger 的选型策略决定了其智能合约的表达能力较为强大（图灵完备），相应地就要对其局限性（停机问题的不可判定性）做好防范准备。

考虑到原生数字货币代码涉及面较广，内在逻辑比较复杂，在代码级别去除原生数字货币激励功能的同时保留资源控制功能，可能会隐含较大的风险。因此，ChinaLedger 在其分布式总账平台的设计中将采用较为温和的方案，保留代码，通过参数设置的方式使原生数字货币与资源控制功能脱钩。

此外，ChinaLedger 在面向资本市场的应用场景中，将有可能会按需引入锚定法币的代币充当“结算币”。它将有助于提升面向实时逐笔结算和交易后清算交收的处理效率，甚至可助力建设数字社区生态。

在未来自主研发版本的分布式总账技术平台中，ChinaLedger 的联盟链版本将不再包含原生数字货币及相关的激励机制。

B.8 性能优化目标

根据国内资本市场的常态和峰值数据，ChinaLedger 将持续优化平台性能，希望最终能达到如下目标。

1）吞吐率目标：每秒 100 000 笔以上。

2）时延目标：同城 1 毫秒以内。

3）存储目标：按每日 80 000 000 笔的容量，重节点能存储全量数据，轻节点能存储当日数据。

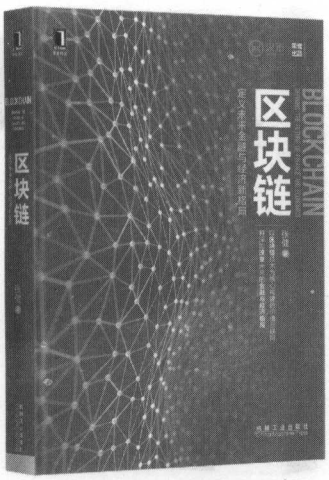
这些优化目标，在初期仅针对场外市场阶段是没有必要的。达到这些目标，主要是为了验证 ChinaLedger 后续支持场内业务交易后业务处理，以及进一步推进分布式总账在资本市场应用的需要。

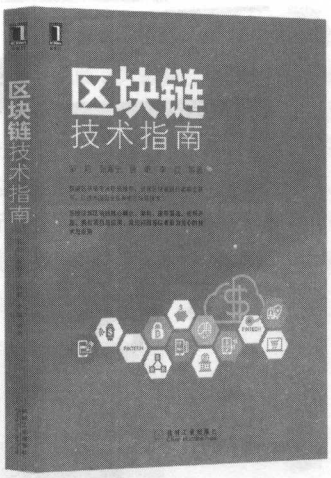
B.9 展望与总结

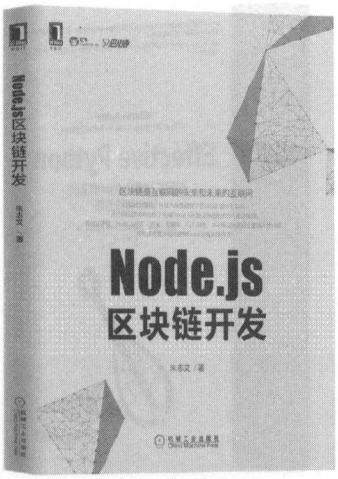
本白皮书仅代表 ChinaLedger 在现阶段对自己的使命、对在资本市场应用分布式总账技术的认识，以及对今后一段时间内平台开发工作的设想。随着项目的推进，我们的认识将会逐步深化，一些观点可能会有所修改，一些方案可能会有所调整和完善。

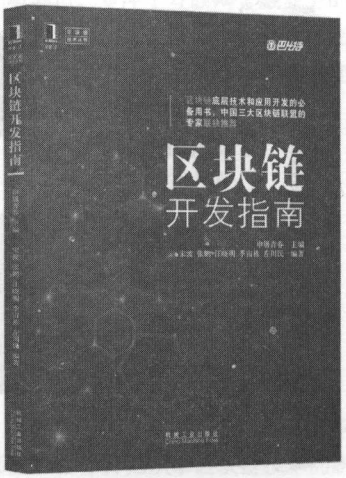
分布式总账技术是涉及强安全性的技术，资本市场又是国民经济的命脉所在，因此自主研发高水平的分布式总账技术平台，是 ChinaLedger 全体成员单位包括观察员单位的共识。同时，我们也了解到，关于分布式总账技术平台的总体思路，跟国内其他的一些联盟组织也是高度一致的。我们将不辱使命，在凝聚共识、整合资源、求真务实、开放开源的基础上，逐步加大自主创新的步伐，做好 ChinaLedger 的基础平台。

推荐阅读

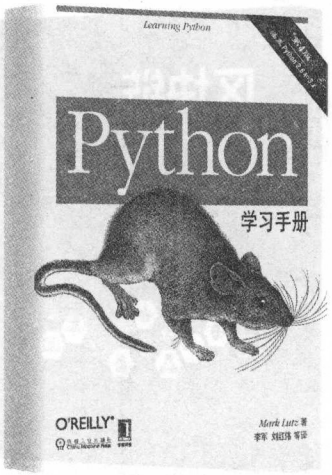

BLOCKCHAIN
区块链
区块链


区块链
技术指南


Node.js区块链开发
Node.js
区块链开发

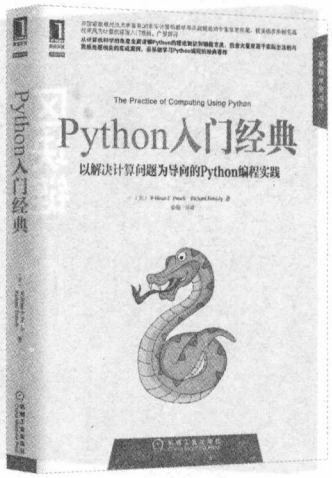

区块链开发指南
区块链
开发指南

推荐阅读



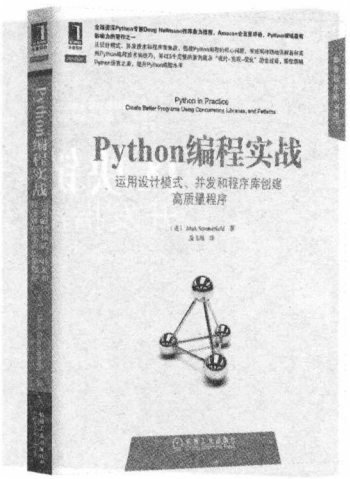
Python学习手册（原书第4版）

作者：Mark Lutz ISBN：978-7-111-32653-3 定价：119.00元



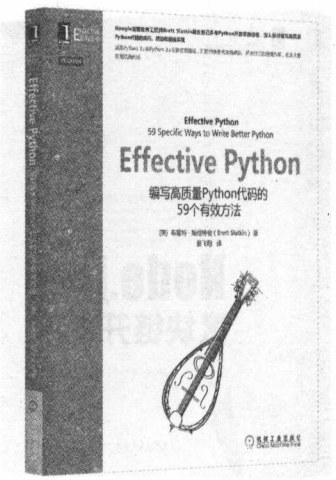
Python入门经典：以解决计算问题为导向的Python编程实践

作者：William F. Punch ISBN：978-7-111-39413-6 定价：79.00元



Python编程实战：运用设计模式、并发和程序库创建高质量程序

作者：Mark Summerfield ISBN：978-7-111-47394-7 定价：69.00元



Effective Python：编写高质量Python代码的59个有效方法

作者：Brett Slatkin ISBN：978-7-111-52355-0 定价：59.00元

汪晓明

朝夕网络 CEO，10 年互联网技术产品经验，在跨境电商、大数据、区块链等领域有丰富的经验。作为区块链技术早期探索者，一直积极推动区块链技术在国内的传播和应用落地，已推出面向金融机构的区块链数字资产和供应链产品。同时发起了有行业影响力的区块链视频节目《明说》，持续影响着更多人参与到区块链技术的研究和推广中。

季宙栋

万达网络科技集团先进技术研究中心副总经理，万达区块链负责人，（工信部）中国区块链技术与产业发展论坛副秘书长，超级账本中国技术工作组委员。专注于互联网金融业务的创新及金融科技实践，参与了工信部区块链白皮书及相关标准编制工作，ISO/IEC TC307 中国代表团成员，牵头工信部区块链开源社区工作。

左川民

先后在多家知名企业担任技术专家和高级架构师。2015 年开始从事区块链技术的研究工作，曾负责积分区块链项目区块链架构设计，专注于区块链技术框架 Fabric 的技术研究，目前为深圳科协、金链盟等机构的 Fabric 技术框架培训讲师，同时作为中国首批区块链系统功能测试评审专家参与区块链系统的评审。